A language capable of describing computer architecture.

Machado, Luis Manuel da Cunha de Sousa

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

A LANGUAGE CAPABLE OF
DESCRIBING COMPUTER ARCHITECTURE

by

Luis Manuel da Cunha de Sousa Machado

March 1986

Thesis Advisor: Hariett B. Rigas

Approved for public release; distribution is unlimited.

URITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

REPORT SECURITY CLASSIFICATION	1b. RESTRICTIVE MARKINGS
SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
DECLASSIFICATION / DOWNGRADING SCHEDULE	
PERFORMING ORGANIZATION REPORT NUMBER(S)	5 MONITORING ORGANIZATION REPORT NUMBER(S)
NAME OF PERFORMING ORGANIZATION: val Postgraduate School / 6b OFFICE SYMBOL (If applicable): Code 62	7a. NAME OF MONITORING ORGANIZATION: Naval Postgraduate School
ADDRESS (City, State, and ZIP Code): nterey, California 93943-5000	7b. ADDRESS (City, State, and ZIP Code): Monterey, California 93943-5000
NAME OF FUNDING / SPONSORING ORGANIZATION / 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
ADDRESS (City, State, and ZIP Code)	10 SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO / PROJECT NO / TASK NO / WORK UNIT ACCESSION NO

TITLE (Include Security Classification)
LANGUAGE CAPABLE OF DESCRIBING COMPUTER ARCHITECTURE

PERSONAL AUTHOR(S)
chado, Luis Manuel da Cunha de Sousa

TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
ineer's thesis	FROM _____ TO _____	1986 March	109

SUPPLEMENTARY NOTATION

COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	computer-aided design tools, top-down design, computer architecture

ABSTRACT (Continue on reverse if necessary and identify by block number)

A fully automated and effective aid for computer system design is of eat interest in increasing designers efficiency and reduce costs. Such system, which requires unified and compatible tools for designing and alyzing computer architectures is still missing. Inserted in a research ogram by Professor Rigas to develop a complete automated design system, is work focuses on designing a formal language capable of describing e data flow of a computer. The language is capable of describing the terconnections between the major data flow components and the control

DISTRIBUTION / AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION Unclassified
NAME OF RESPONSIBLE INDIVIDUAL ariett B. Rigas	22b TELEPHONE (Include Area Code) (408) 646-2082 / 22c OFFICE SYMBOL Code 62

FORM 1473, 84 MAR 83 APR edition may be used until exhausted
All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

19. ABSTRACT (cont'd)
of the flow of information. Using this language, several decompositions of the intended system can be specified and studied to find the optimal one.

A Language Capable of Describing Computer Architecture

by

Luis Manuel da Cunha de Sousa Machado
Lieutenant, Portuguese Navy
B.S., Escola Naval, 1978

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

and

ELECTRICAL ENGINEER

from the

NAVAL POSTGRADUATE SCHOOL
March 1986

ABSTRACT

A fully automated and effective aid for computer system design is of great interest in increasing designers efficiency and reduce costs. Such a system, which requires unified and compatible tools for designing and analyzing computer architectures is still missing. Inserted in a research program by Professor Rigas to develop a complete automated design system, this work focuses on designing a formal language capable of describing the data flow of a computer. The language is capable of describing the interconnections between the major data flow components and the control of the flow of information. Using this language, several decompositions of the intended system can be specified and studied to find the optimal one.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I wish to express my sincere appreciation to Professor Rigas for her guidance and assistance during the pursuit of this study.

To my wife and my son for the encouragement, patience and understanding I am deeply grateful and it is to them that I dedicate this work.

I. INTRODUCTION

A. THE DIGITAL SYSTEM DESIGN PROCESS

The major factor influencing digital system design has been the rapid evolution of semiconductor technology. Whereas in the recent past, digital systems employed vacuum tube circuitry, today digital systems are built from IC chips containing ten thousand or more gates, and the number of gates per chip will grow in the near future. Digital systems are complex structures of many gates. To deal with such complexity, digital architects decompose these structures into functional blocks aggregating a number of gates performing a well defined task. Two or more of these blocks may form, in turn, a single but higher abstract block. For the architect, each of these blocks is characterized by the function it performs, the interface with other blocks, and the time taken to perform its function. This means that a digital system can be described at several levels of abstraction.

The use of these abstract layers, allows a hierarchical approach to the design process. Starting from a set of specifications and applying a series of successive expansions, the architect steps through the different levels of abstraction in a top down fashion until the physical design can be implemented using available technology.

In summary, design of large digital systems is a complex, costly, and time consuming process. Therefore, the development of automatic design aids to overcome these drawbacks is of great interest.

B. THE ROLE OF DESIGN AUTOMATION

Design automation can be defined as the application of today's computers to the design of tomorrow's computers. The

major functions of design automation can be summarized as follows:

- Replace the designer in tasks that are well understood and where no decisions are to be made.
- Assist the designer in making decisions by evaluating the merits of various design alternatives.
- Assist the designer in verifying the correctness of his design.

It is not apparent that in the near future humans will be fully replaced by machines in the design process. Therefore, future systems will be the result of joint work of humans and computers. A CAD system should then make use of the best attributes of the computers (record keeping, searching and massive computational capabilities) and the best attributes of the user (pattern recognition and rational thought).

Recently, a great deal of effort has been directed to the development of techniques and tools for allowing computers to perform the tasks described above. Unfortunatly, the design aids were developed independently as the need for them arose, causing the following problems:

- lack of compatibility among the various design automation tools.
- lack of extensibility to firmware and software.
- usage complexity.
- poor interaction with the designer/user.
- lack of unified database to provide consistency checking through-out the design process.

Consequently, an efficient automated design system is still missing.

C. A PROPOSED COMPLETE AUTOMATED DESIGN SYSTEM

Figure 1.1 shows a model for a complete automated design process proposed by Professor Rigas. [Ref. 1]

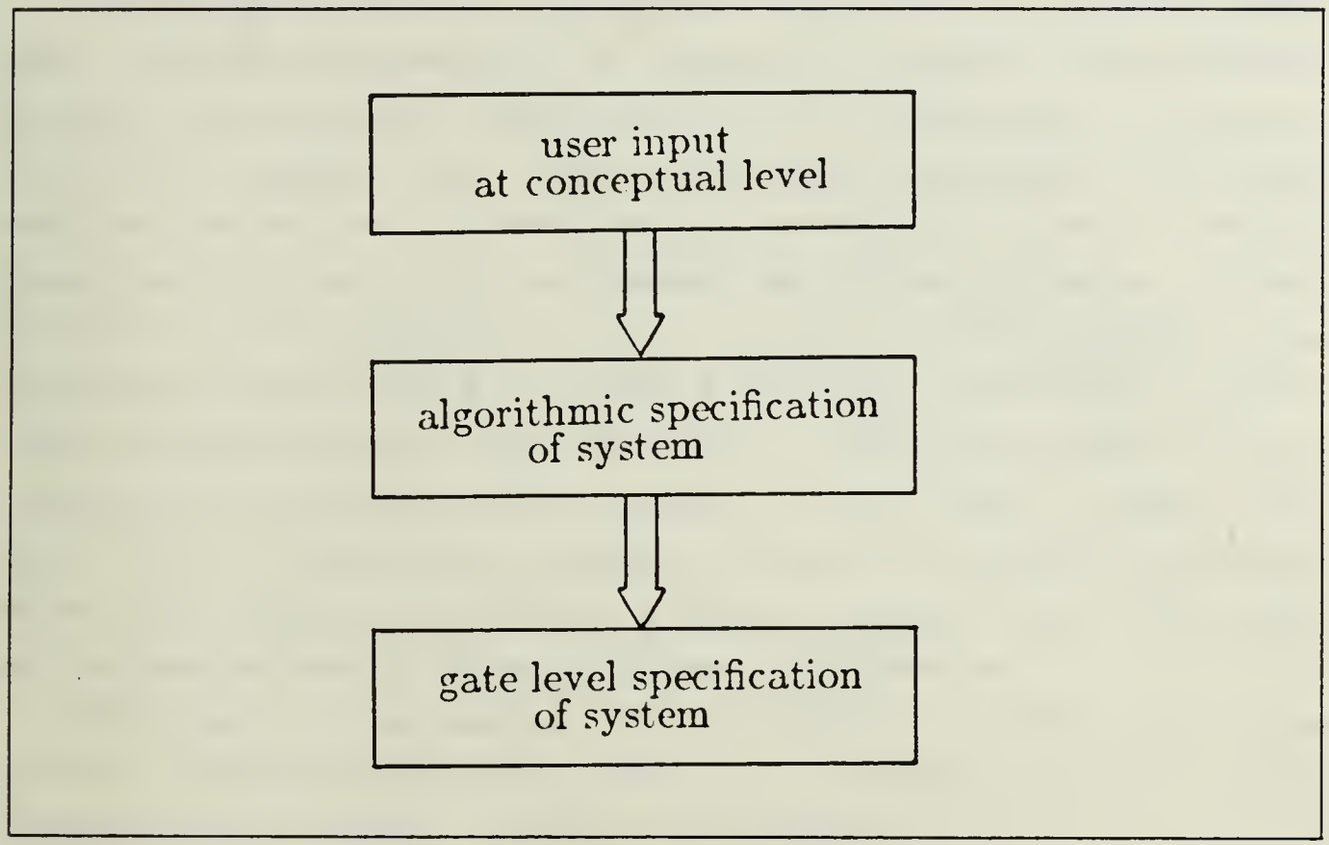


Figure 1.1 A Complete Automated Design System.

The upper portion of Figure 1.1 concentrates on generating a high-level hardware/software description of the system from a description of the problem to be solved.

The middle portion focuses on generating a design language suitable for describing the flow diagram of a system. Using this language, several decompositions of the intended system can be specified and studied to find the optimal decomposition.

The lower part concentrates on the gate implementation of the system. A hardware description language and an event driven-driven simulator capable of analyzing hardware performance at the gate-leval have been developed and tested. [Ref. 2]

This work focuses on the middle portion of Figure 1.1 Its goal is to design a formal language capable of describing the data flow of a computer system.

II. BACKGROUND

A digital system is any interconnection of digital hardware modules assembled to process digital information. Digital information simply means that information is represented by signals that take on a discrete number of values and is processed internally by components that normally function only in a limited number of discrete states. State refers to "the property of a machine which relates the inputs to the outputs in such a way that knowledge of the input time function f(t) for $t \geq t_0$ and the state at $t=t_0$ completely determines the output for all $t \geq t_0$ " [Ref. 3:p. 292]. For reliability purposes, digital systems use components that take two discrete states, meaning that the information processed by them is binary information.

The major components of a computer system are the Central Processing Unit (CPU), the Memory Unit and the Input/output devices, as shown in Figure 2.1.

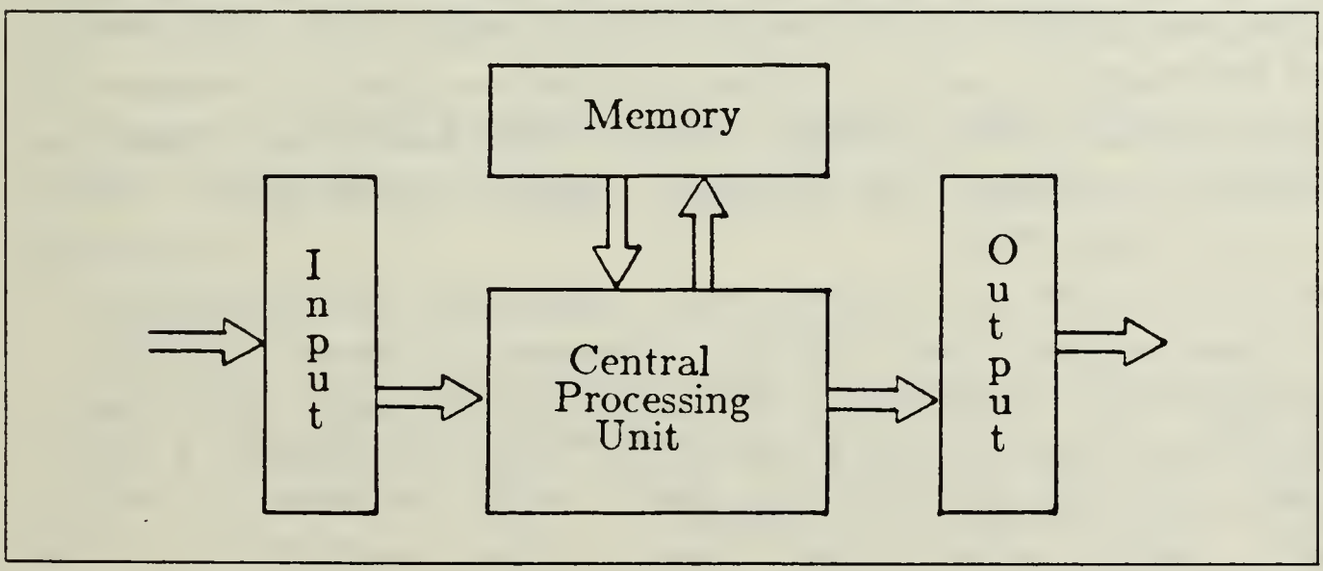


Figure 2.1 Block Diagram for a Digital System.

To accomplish its task, a computer must be supplied with information to be processed (the DATA), and with information to guide it in performing its work (the CONTROL). The Central Processing Unit of Figure 2.1 can be conceptually broken into two parts as shown in Figure 2.2.

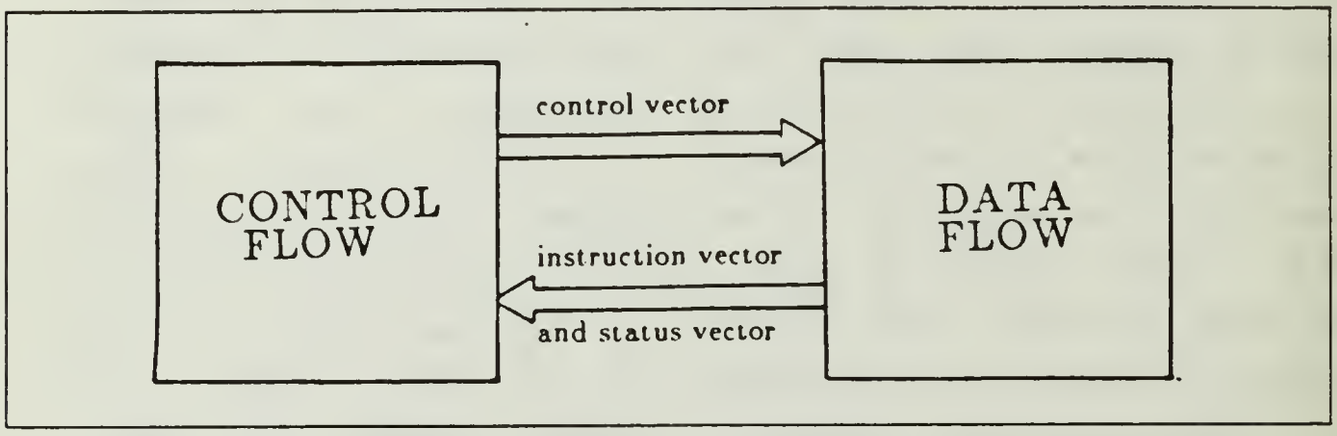


Figure 2.2 The DATA FLOW and CONTROL FLOW Components.

The DATA FLOW (or DATA PATH) component is the one that manipulates the data. The DATA FLOW is capable of accepting, processing and delivering data. The building blocks comprising the DATA FLOW are called DATA FLOW components. They are the Memory Block and the Functional Unit interconnected by buses.

The CONTROL FLOW (CONTROL PATH, CONTROL UNIT or simply CONTROLLER) provides the control signals which guide the data in the DATA FLOW.

The DATA FLOW and CONTROL UNIT components are physically built from digital logic blocks, elements or gates. Digital logic gate-level blocks are the primitive or basic decision elements such as AND and OR gates, and primitive 1-bit memory elements called flip-flops.

A. BINARY VECTORS

Binary information in digital systems is stored in memory devices. A memory device consists of a number of storage cells, each of which can store a binary digit, or

bit. Since one bit represents only a very small amount of information (it can only have the value 0 or 1), bits are seldom handled individually; instead they are handled in vectors. A vector of n-bits, which together convey an item of information, is called a word, and n is called the word length.

The information in a word is obtained by assigning specific weights to the bit positions. The bit with the least weight is the Least Significant Bit (LSB) and the bit with the most weight is the Most Significant Bit (MSB). The bits within the word are depicted from the left to right, bit 0 through bit n-1 if the MSB 0 numbering convention is adopted or from right to left if the LSB 0 scheme is in used. The latter is the one adopted throughout this work. Thus, the binary vector V<3:7> has length five, its least significant bit is bit 3 and its most significant bit is bit 7, as shown below:

$$V = (V_7, V_6, V_5, V_4, V_3)$$

Words may be used to stand for data or control. Control information is a string of bits used to specify the sequence of command signals needed for manipulation of the data in the Data Flow. Data are binary numbers and other binary-coded information that are operated on by the Data Flow components.

B. BUSES

To reduce the number of wires necessary to comprehensively interconnect system devices, buses are used. A BUS is a parallel group of wires, grouped together because of similarity of function, which connect two or more devices. The devices that have their outputs connected to the bus are the SOURCES of the bus, and devices that have their inputs attached to the bus are the SINKS. In Figure 2.3 a), S is a source for the TBUS and A,B,C and D are the sinks. The

number of wires grouped together determines the WIDTH of the bus, e.g., the maximum length of the binary vector that can flow through it. These signal wires are used to transfer data from a source to one or more destinations.

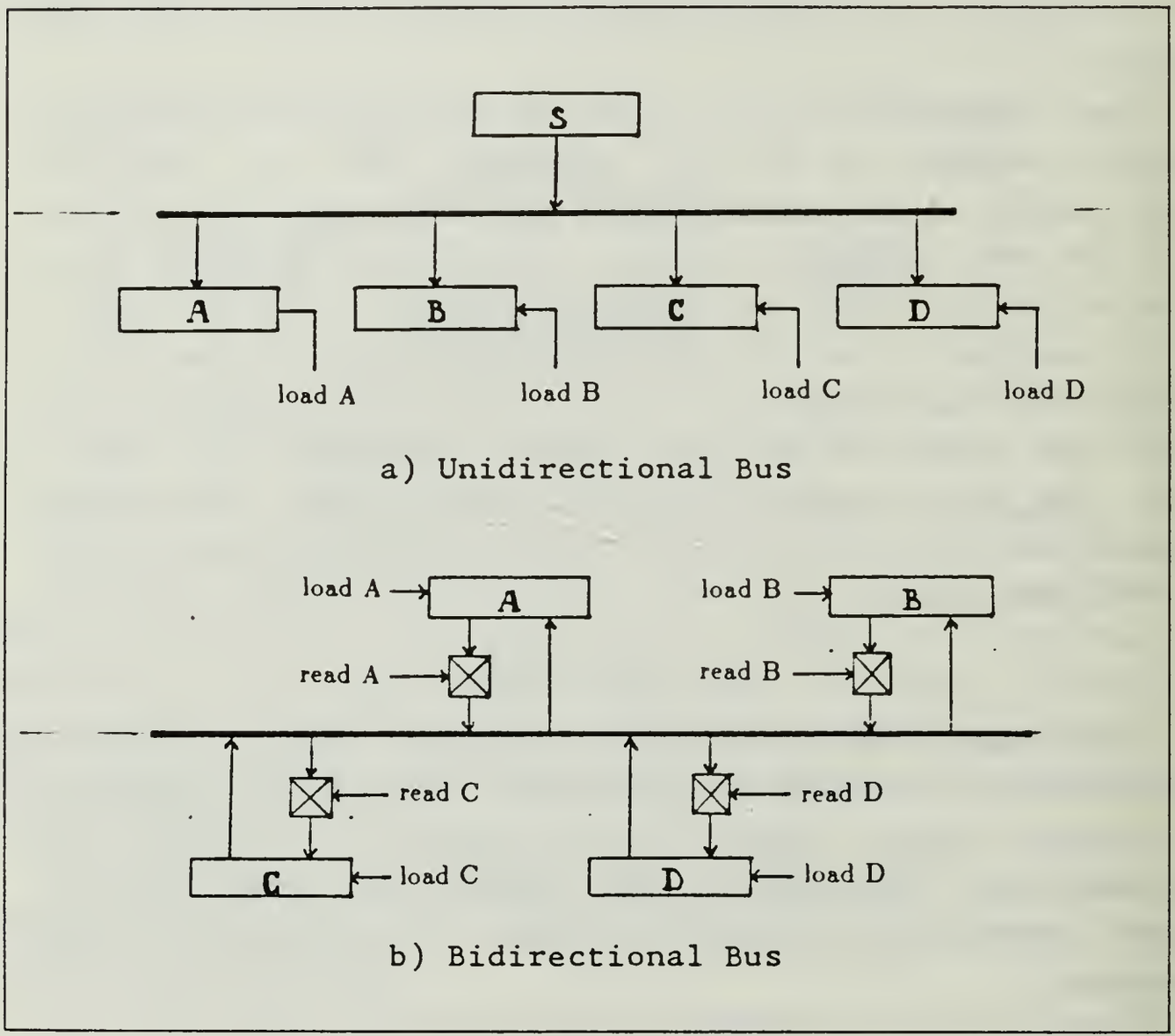


Figure 2.3 A Bus System.

Buses may be of two kinds:

- Unidirectional buses.
- Bidirectional buses.

An unidirectional bus is a bus that have only one source. A bidirectional bus is a bus that can have more than one source, but not more than one source can be active at the same time. This being the case, coordination logic at each possible source is necessary, in order to avoid more than one device driving the bus at the same time.

The techniques used to accomplish this are:

- Using multiplexers. [Ref. 4]
- Open-collector. [Refs. 5,6]
- Tri-state drivers. [Refs. 5,6]
- The transmission gate. [Refs. 7,8]

The transfer of information from a bus into one of many possible destinations is accomplished by connecting the bus lines to the inputs of all destination devices and enabling the particular device selected by activating its load control signal. Figure 2.3 shows an unidirectional bus with four destinations and a bidirectional bus connecting four devices.

To reduce the number of control lines, the LOAD and READ are generally encoded, as illustrated in Figure 2.4.

The inputs to the decoder represent the address of the device for which the READ or LOAD signal is to operate on.

C. FUNCTIONAL UNIT

A FUNCTIONAL UNIT is a combinational logic device which accepts one or two n-bit input vectors and generates an output function $S = S_{n-1},\ldots,S_0$ which is related to the inputs by boolean logic.

Because an arithmetic binary operation assigns a binary vector for all possible combinations of the input vectors, it can be described by a truth table. A truth table is way for describing the behavior of a combinational circuit. As a result, arithmetic binary operations can be physically

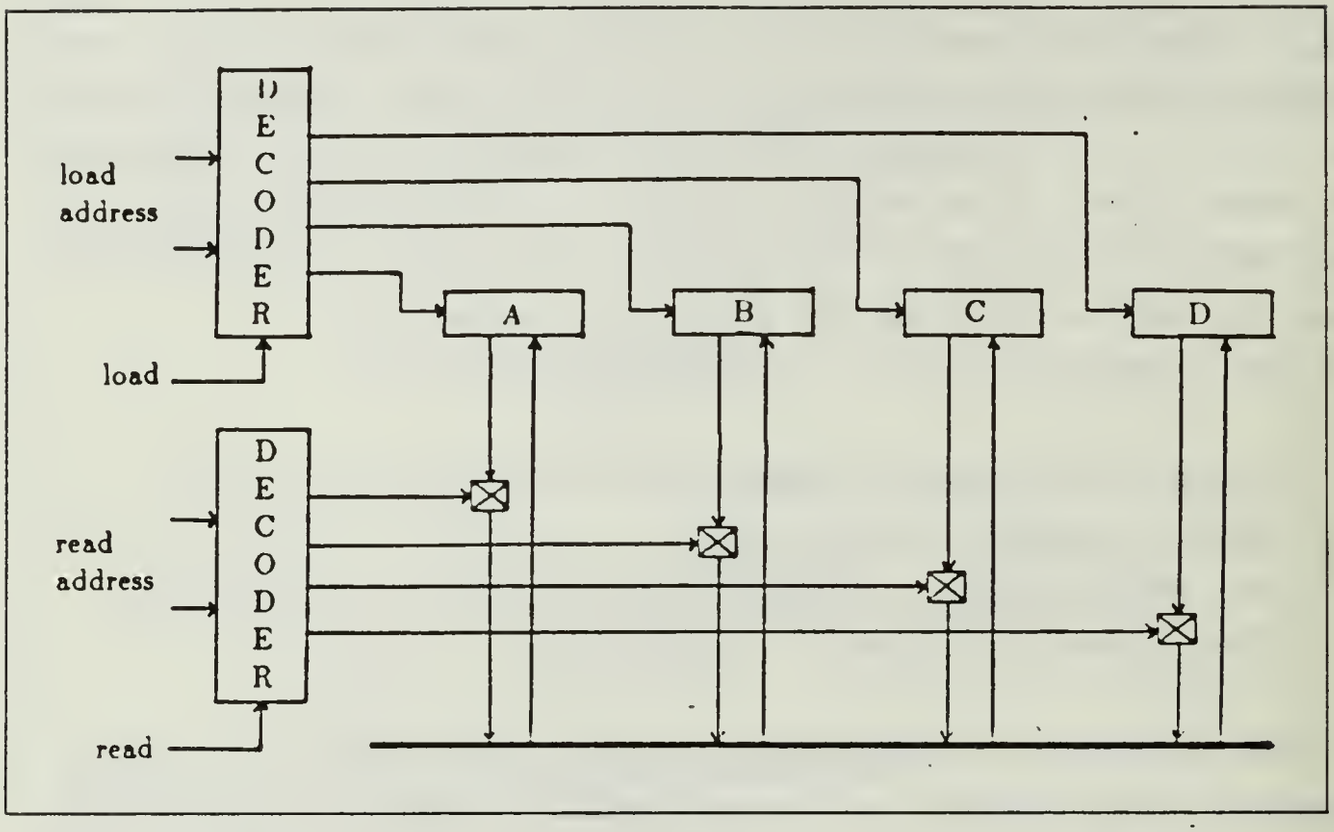


Figure 2.4 The Use of a Decoder to Save Control Lines.

realized by combinational circuits. This means that the Functional Unit output vector may represent the result of either a logic or an arithmetic operation on its inputs.

An operation-selection vector determines what specific function is to be generated. Additionally, the functional unit may provide a status vector containing information about the output (overflow, zero, carry out, parity, etc.). Figure 2.5 shows the block diagram for a functional unit.

Each of the operations performed by the functional unit corresponds to a functional path between the inputs and the output. By activating the proper bits in the operation-selection vector, it is possible to select a particular path, thus choosing a particular operation to be performed.

The Functional Unit is combinational logic because its output does not depend on the past history of the device, but rather is strictly a function of its inputs. The Functional Unit has no memory. Its operation is not invoked

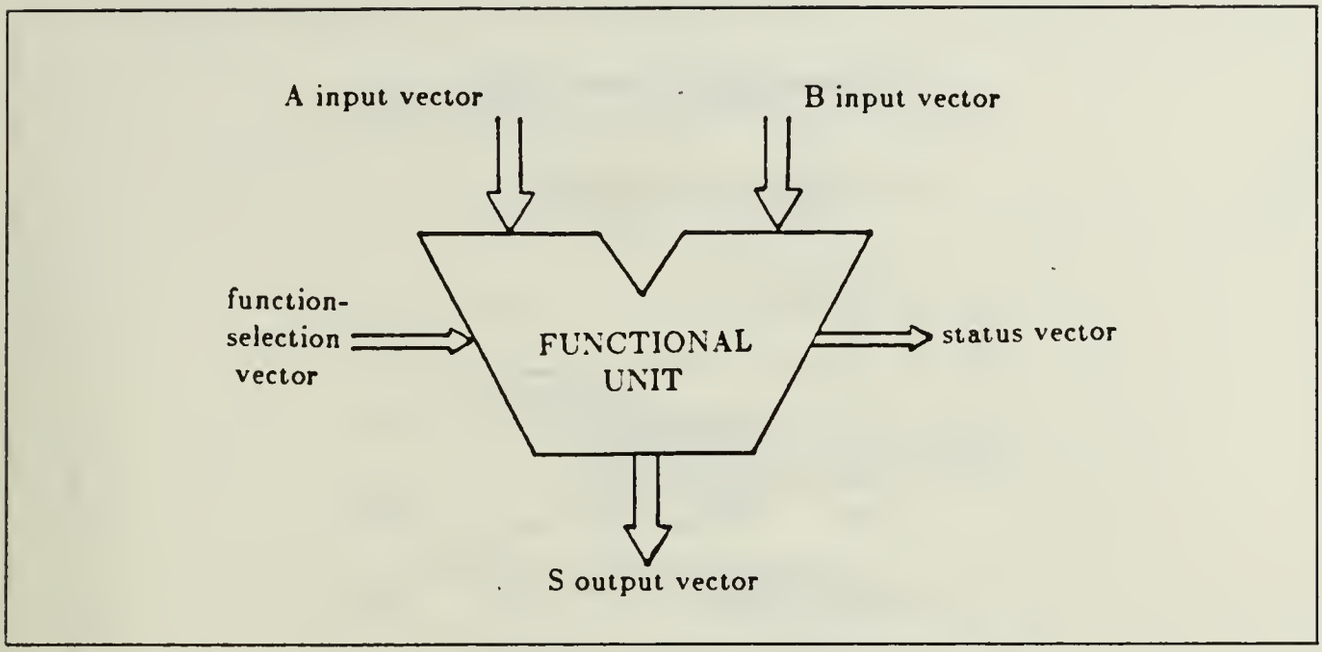


Figure 2.5 The Functional Unit Block Diagram.

by a clock. The only timing consideration is that the output validity is subject to propagation delays.

When the inputs of a logic gate change, the output of the logic gate output does not respond instantaneously to the change. A propagation delay must be paid before the output stabilizes to the new value as is illustrated in Figure 2.6 for an inverter element.

The fall and rise delays are not equal nor fixed. The fall and rise delays depend on such factors as temperature and fan-out. Because of these variations, logic designers use Worst-case Propagation Delays (the maximum possible propagation delays), Best-case Propagation delays (the minimum possible propagation delay) and Average Delays. The period of time between the worst-case delay and the best-case delay is called the Ambiguity Region, sometimes refered to as Propagation Skew.

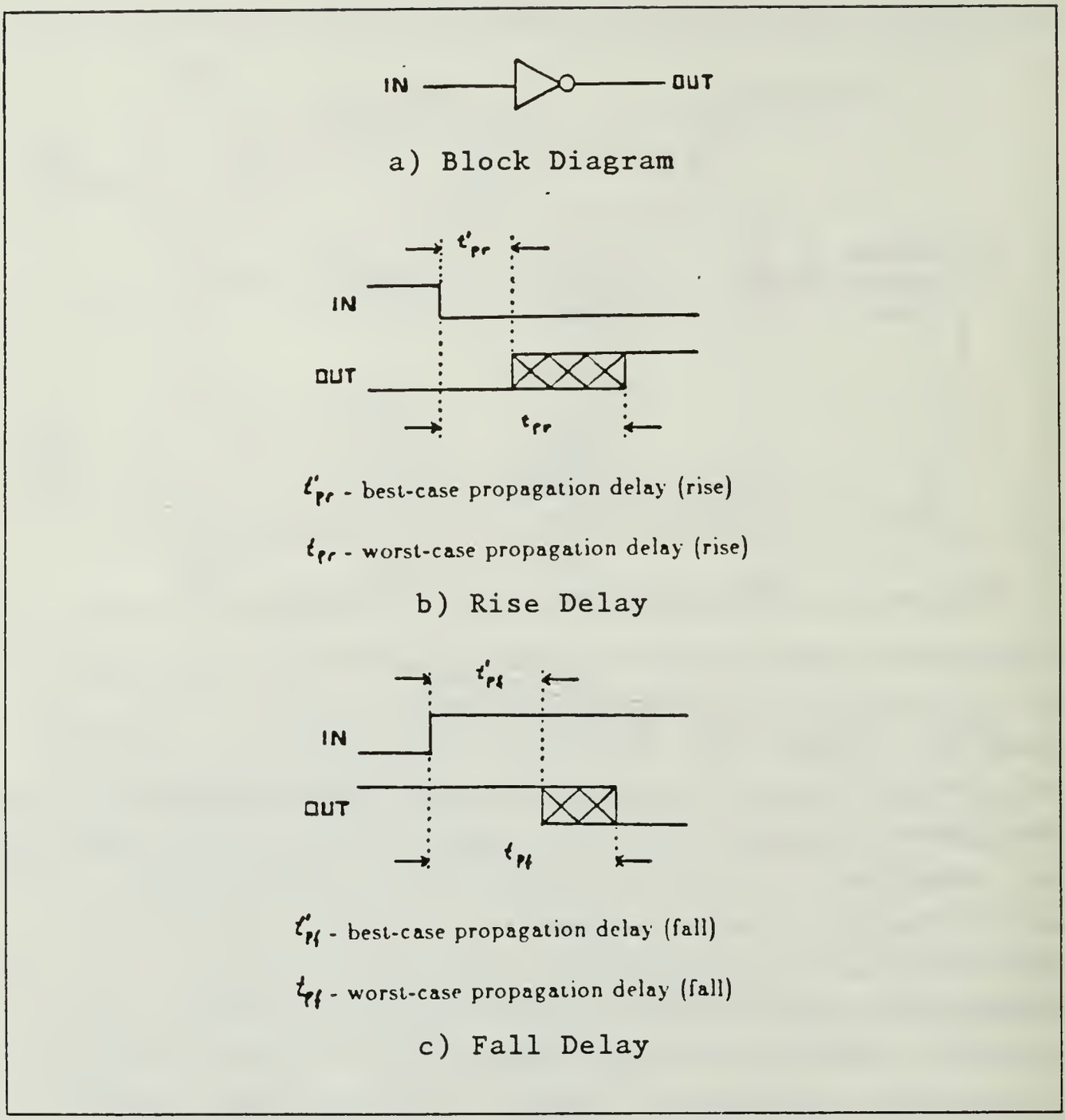


Figure 2.6 Timing Diagrams for a Simple Inverter.

It is not uncommon to estimate the propagation delay of a cascade of logic gates as the sum of the individual gate delays as shown in Figure 2.7.

Each Functional Path is a cascade of logic gates. Therefore, each one has a propagation delay. For the system point of view, the Functional Unit propagation delay is the

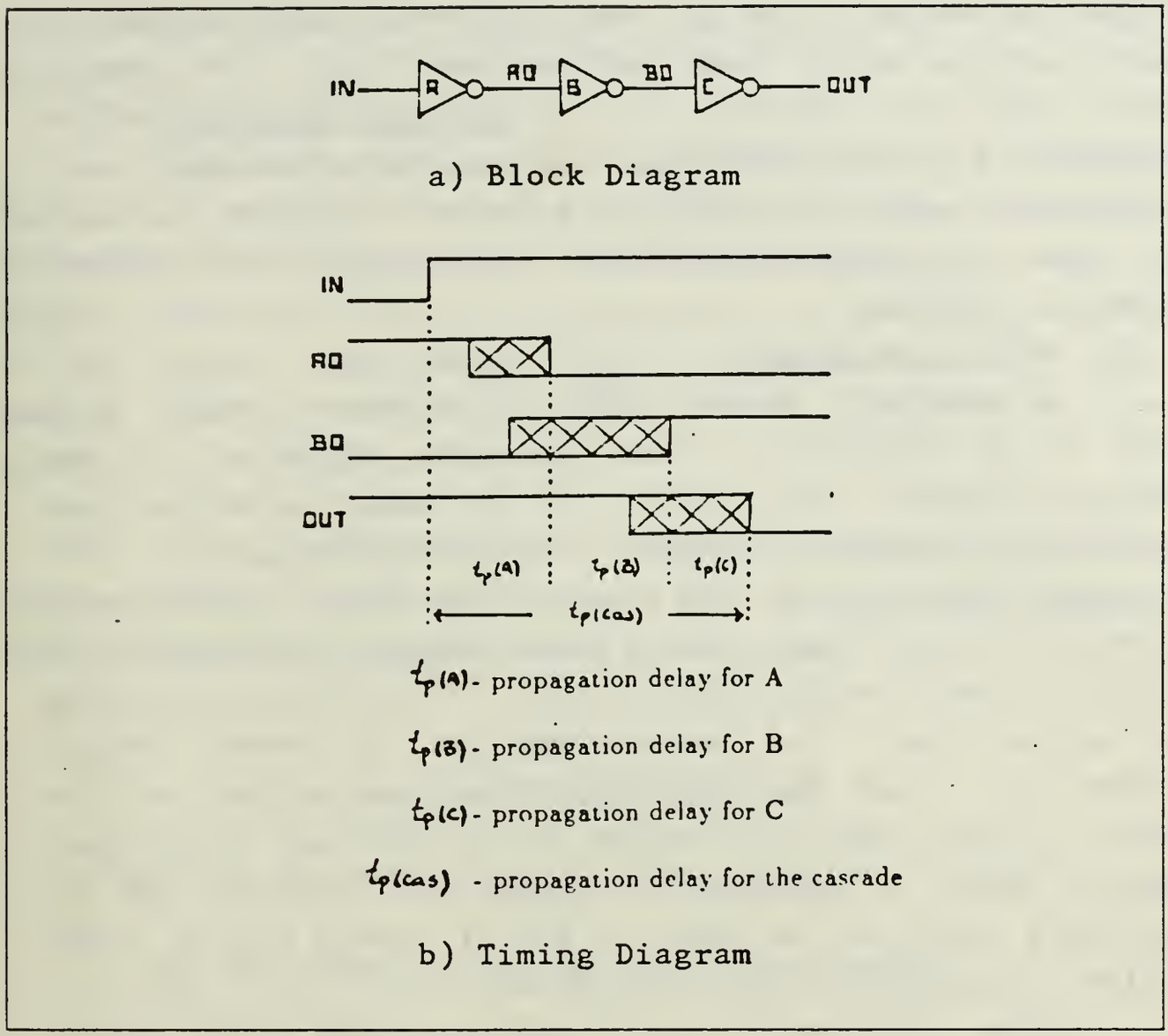


Figure 2.7 A Cascade of Inverters.

maximum of the propagation delays of its paths, e.g., it is the time necessary for a change in its inputs to propagate to the output through the slowest path.

D. MEMORY DEVICES

Memory devices are devices capable of storing information. With a variety of such devices, such as magnetic disks, tapes, bubble memories, RAM's, ROM's and registers, this work only contemplates those made from semiconductor devices, which are the memory devices found in the Data Flow of a system. Two major categories, based on construction,

of devices exist. The following subsection will discuss memory devices built from combinational logic. The memory devices that are sequential circuits will be analyzed in subsection 2. The information in the system is as binary vectors or words each of which is stored in some location that can be referenced through an identification number called the ADDRESS.

1. ROM's and PLA's

A read-only memory (ROM) is a memory device from which it is possible to read but into which it is not possible to write. The contents of the memory are fixed and unalterable. Because a ROM is a combinational circuit, the only time constraint is the propagation delay, in this case called the Access Time. When a K-bit address is presented to a ROM, a stable m-bit output vector is delivered following the access time. The memory contains 2^k words, called p-terms, one used for each possible combination of the address lines. The word length is m bits and a distinct physical word is permanently stored for each of the 2^k distinct p-terms of the ROM. A ROM of size m x n is a ROM storing n binary vectors of length m.

Like a ROM, a PLA has k address lines and m output lines. However, a PLA does not use all possible combinations of its address lines, in other words, it has fewer p-terms than a ROM with the same number of address and output lines. As a result, an important specification for a PLA is the number of p-terms it has; this number represents the number of AND gates available. The outputs of each of these gates can drive or not drive each of m OR gates. The AND gate section of the PLA is called the AND-array and the OR gate section the OR-array. The PLA, then, is a direct way to realize a two level combinational circuit of the AND-OR type. PLA's can be used on Control Units as memory or in the Data Flow as realizations of systematic operations on

data words such as addition, multiplication or sign extension.

2. Registers and RAM's

Registers and RAM's are sequential circuits devices. Sequential circuits are circuits, which involve feedback, and exhibit the feature that the outputs depend not only on present inputs but also, to some extend, on the past history of the inputs. This means that a sequential circuit has memory. What is remembered is stored in a flip-flop, the simplest sequential circuit.

a. Flip-flops

Flip-flops can be classified according to the way they are clocked and to the way they are controlled. Table I shows the categories of flip-flops.

TABLE I
FLIP-FLOP CLASSIFICATION

a) According to the way they are clocked:

1) level-sensitive (data latch)
2) edge-triggered

b) According to the way they are controlled

1) set and clear
2) D type
3) J-K flip-flop (only edge-triggered)

Presently the most widely used flip-flops are of the level-sensitive and edge-triggered types. Figure 2.8 shows the block and timing diagrams for a Data latch.

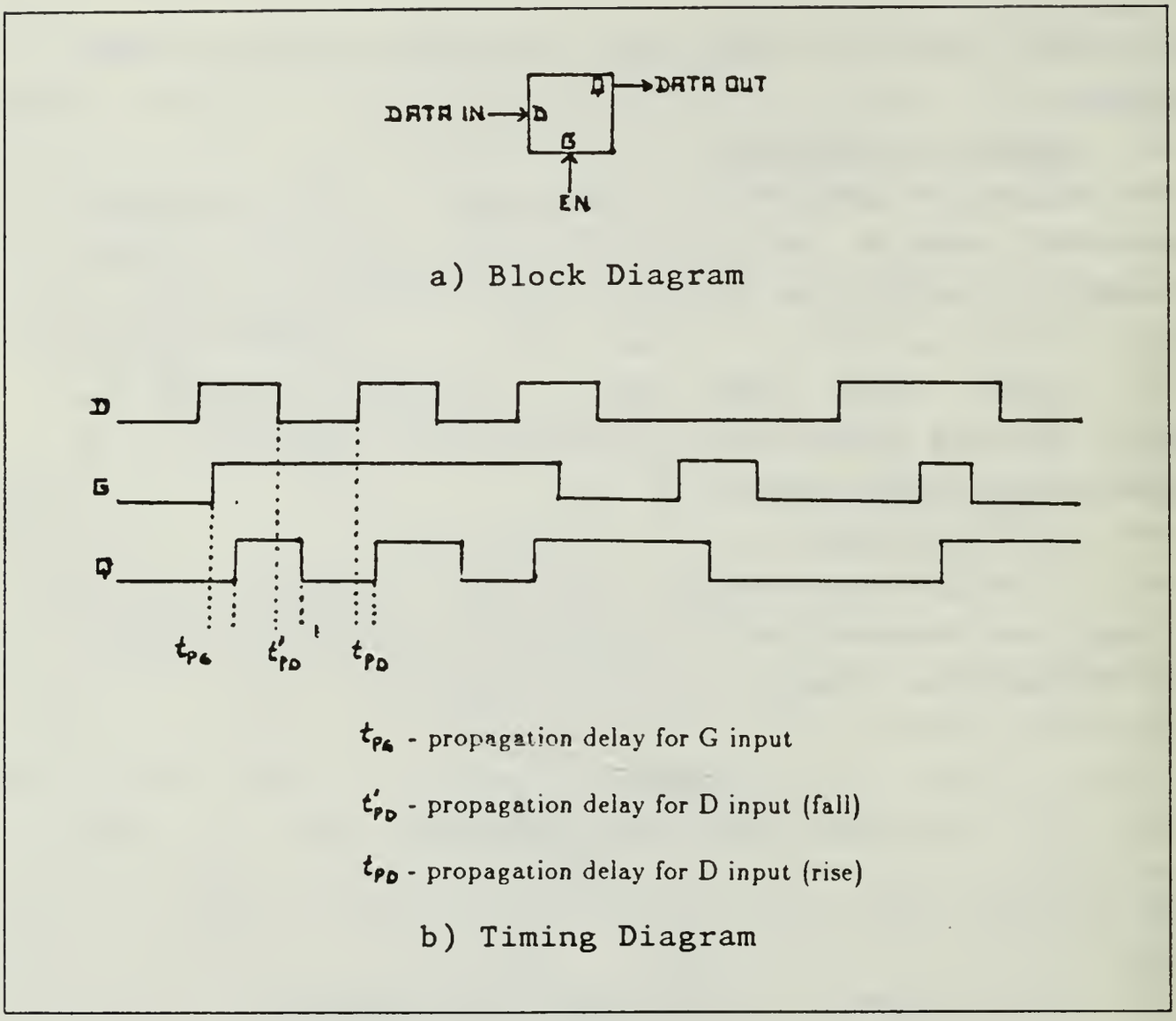


Figure 2.8 The Data Latch.

Note the two different sources of propagation delay: the propagation delay associated with the D input and the propagation delay associated with the G input. Also important is the concept of setup time and hold time. Setup time is the minimum time that input D must be stable prior to the deactivation of G, to guarantee that a known value is latched. The hold time is the time during which data will be steady and valid after control point G has been deactivated.

A block and timing diagrams for a D edge-triggered flip-flop is shown in Figure 2.9. The sampling

interval is the time during which the D input must remain stable to guarantee a correct value at the output and is equal to the sum of the setup time and the hold time. The output hold time is equal to the best-case propagation delay.

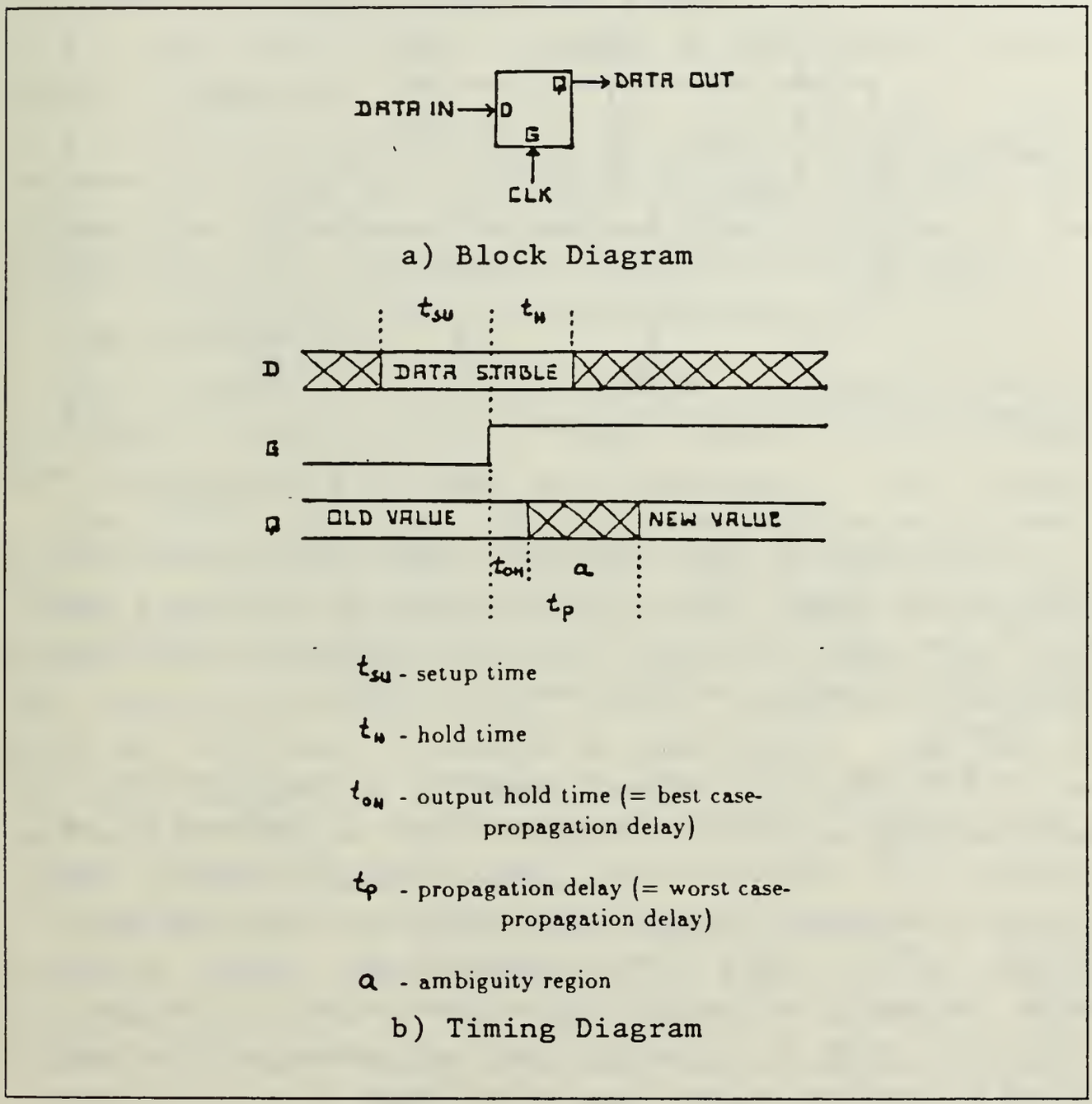


Figure 2.9 The Edge-Triggered Flip-Flop.

3. Registers

Registers are composed of individual flip-flops, usually edge-triggered or data latch, with common control and clocking signals as illustrated in Figure 2.10.

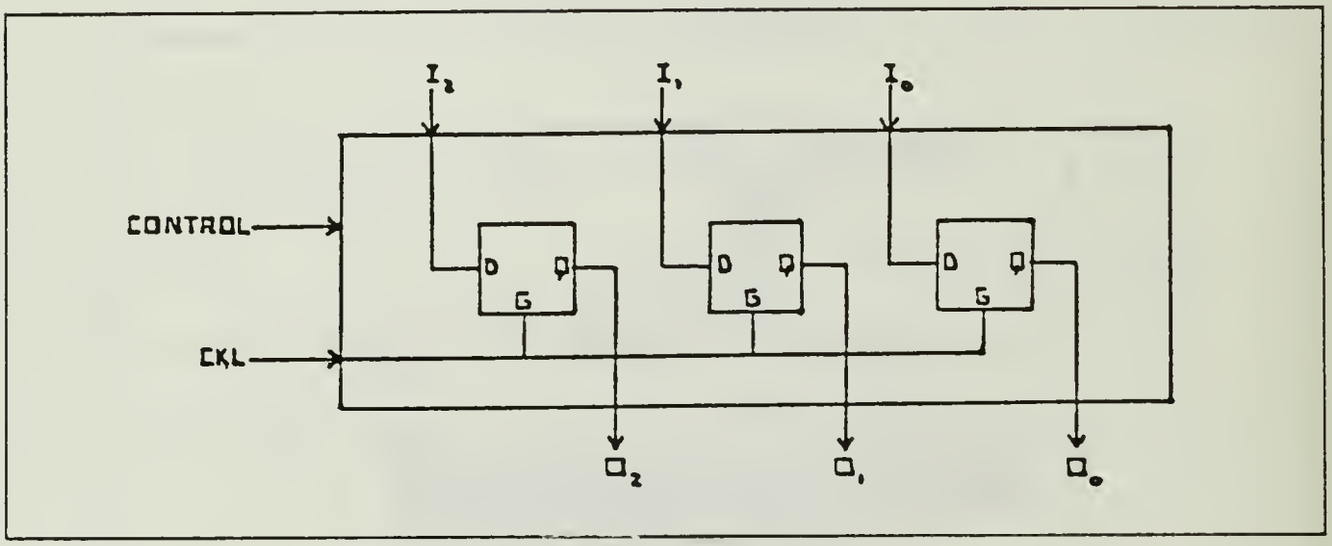


Figure 2.10 A Register as an Array of Flip-Flops.

The number of flip-flops in a register dictates the maximum binary vector that can be stored in it, e.g., the length of the register. For locality purposes each register is identified through a unique name (address).

4. RAM's

A RAM can be thought as an array of registers built from data latches. Each of these registers is refered to by an address which differentiates them within the array. The Ram is two-dimensional, because the length of the individual registers defines the word length while their number dictates the number of words that can be stored in the RAM. The notation "2048 x 4" means that RAM contains 2^K 4-bit words and is refered as the size of the RAM.

In general a RAM cannot perform a READ and a WRITE operation simultaneously and therefore only one control signal is necessary. While the WE (write enable) control signal is not active the contents of the register whose

address is in the address input, is present at the RAM output. By activating WE the value of the data input is loaded into the register. In fact memory chips have an additional control signal, the Chip Select which when activated, enables the RAM to behave as described. Figure 2.11 shows the block and timing diagrams for a RAM.

Note that WE can be thought as the G input for the latch of Figure 2.8.

An important measure of the speed of a RAM, besides the memory access time, is the memory cycle time, e.g., the minimum time delay required between the initiation of two independent memory operations.

E. THE CONTROL UNIT

The behavior of a digital system is characterized by transfers of binary vectors between memory devices through data paths in the DATA FLOW.

The CONTROL UNIT is a finite-state machine whose function is to control these transfers. The Control Unit uses inputs from the system clock to derive timing and control signals which regulate the data transfers associated with each instruction (this is only true for synchronous machines which are the ones considered in this work). The Control Unit also accepts as an input vector the contents of the instruction register and the status vector, and generates an output vector of control signals.

The Control Unit cyclically steps through a finite number of states, the CONTROL STATES. Based on the present state and the value of the input vector, the Control Unit changes to a new state in synchrony with the system clock.

Typically, the Control Unit must stay in a control state for a period of time long enough to allow the slowest data transfer in the Data Flow can take place. The cycle of the Control Unit is called the MACHINE CYCLE and it may comprise one or more control states depending upon the architecture

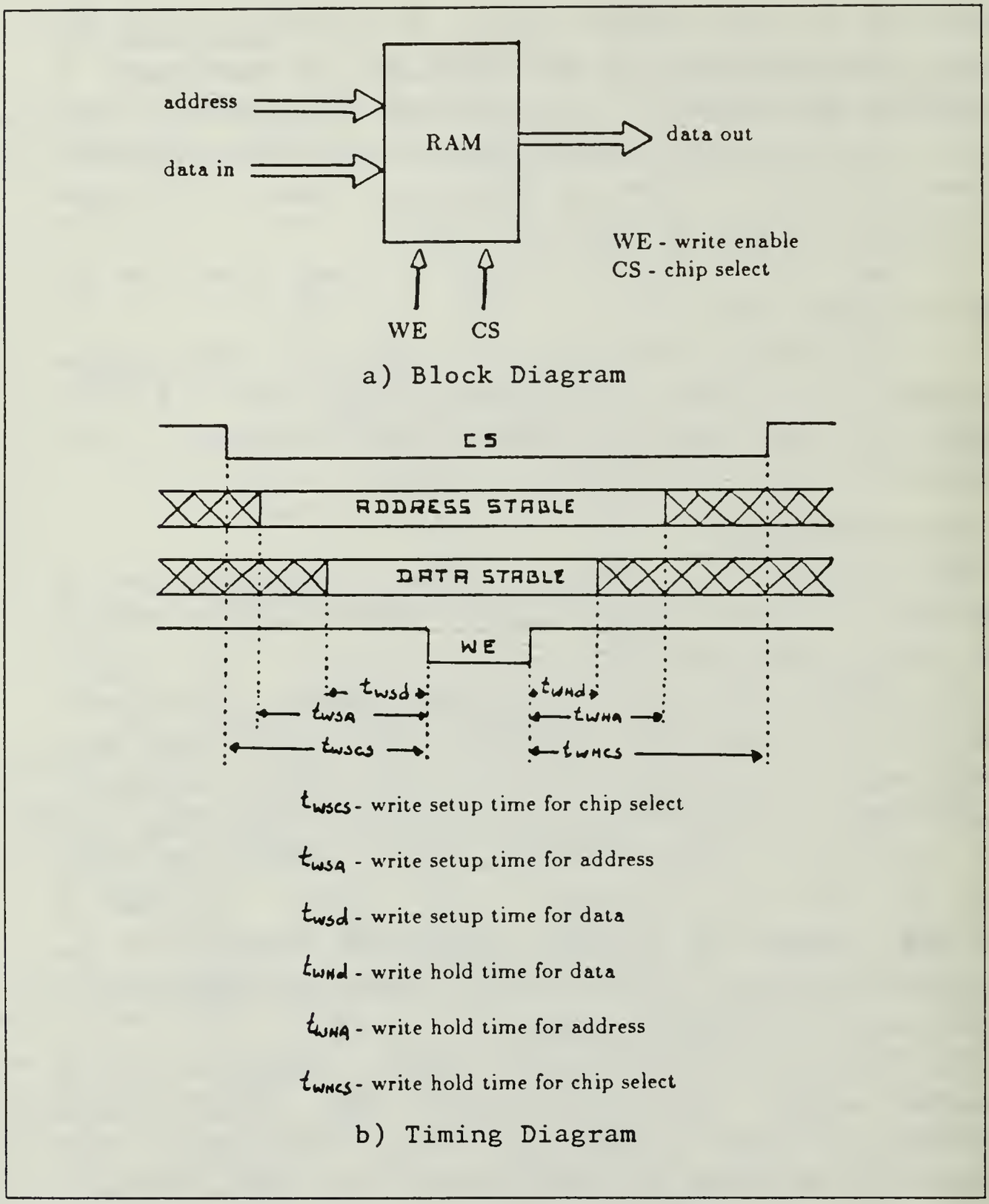


Figure 2.11 The Read-and-Write Memory.

of the system (specially the address modes implemented) and the particular instruction to be executed.

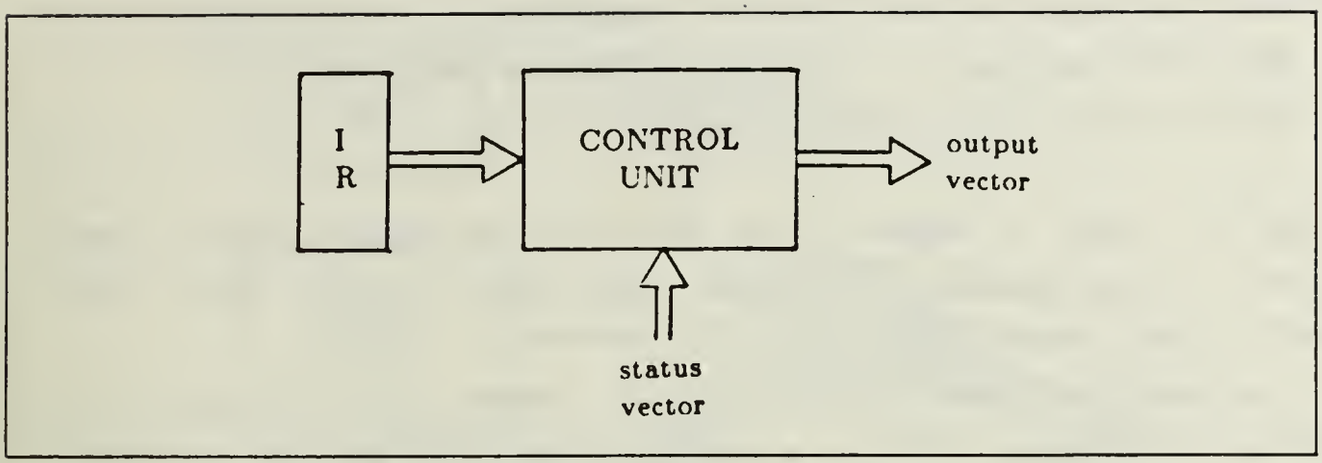


Figure 2.12 The Control Unit Block Diagram.

The Control Unit may be hardwired or it can be implemented using a technique, first presented in 1951 by M. V. Wilkes, called microprogramming.

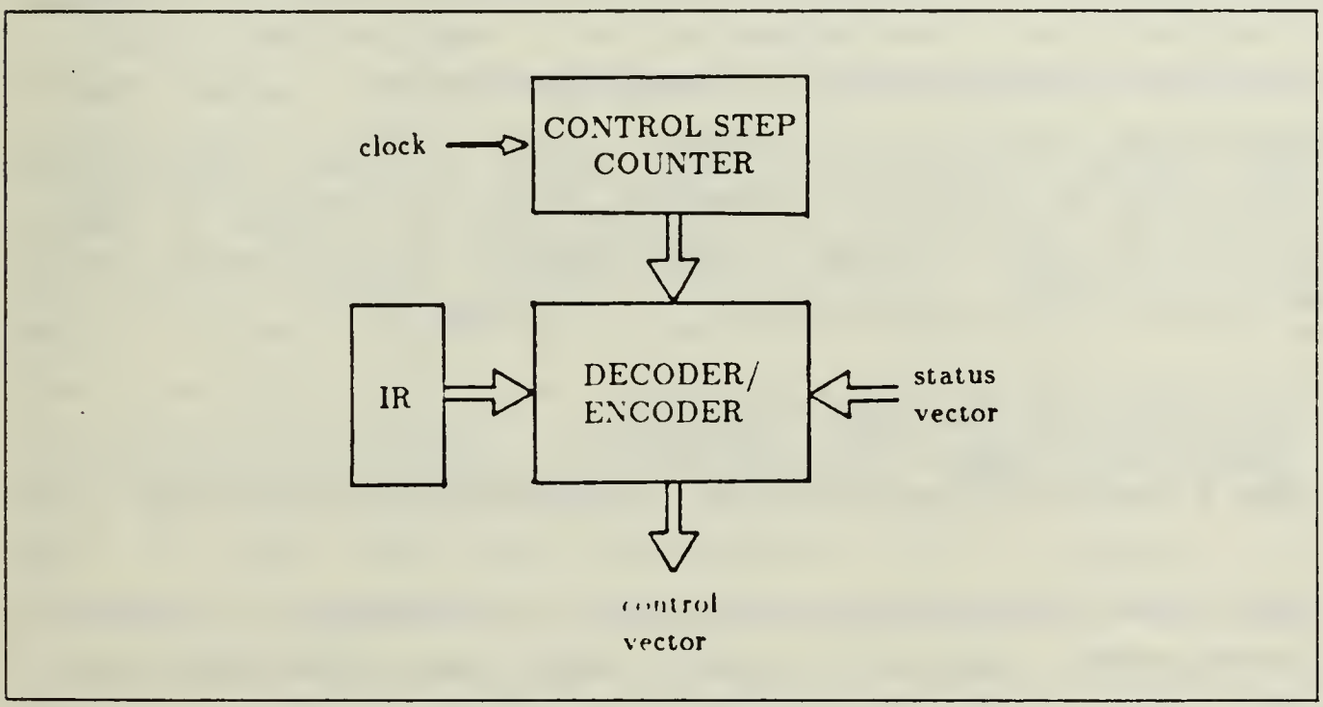


Figure 2.13 A Hardwired Control Unit Block Diagram.

Figure 2.13 shows the hardwired implementation. The decoder-encoder block is simply a combinational circuit that generates the required control vectores, depending upon:

- the contents of the control register.
- the OP code part of the instruction register.
- the value of the status vector.

By OP code is meant "the part of an instruction that specifies the operation to be performed during the next cycle". [Ref. 9:p. 609]

A microprogrammed control unit, whose block diagram is depicted in Figure 2.14 is a control unit having the control vectors stored in a memory (the Control Memory). Each control vector in memory is called a microinstruction and a sequence of microinstructions is called a microprogram. Since alterations of the microprogram are seldom needed, the memory is typically a read-only memory (ROM). A set of microinstructions, specifying a routine, corresponds to each user instruction or macroinstruction. Combinational logic maps the macroinstruction to the ROM address where the corresponding routine is stored. From there, the next microinstruction address, depending upon the value of the status vector and the load control selection bits specified in the present microinstruction, is obtained by:

- incrementing the CMAR register.
- loading the CMAR register with the address specified in the branch address field of present microinstruction.

The hardwired implementation has the advantage of speed and consequently is used in fast, large-scale machines. The latter leads to more versatile controllers because it is usually easier to change a microprogram (software) than to change hardwired logic.

F. SOME NOTATION

As was already stated, much of the activity of a digital system consists of operating on data and transfering vectors

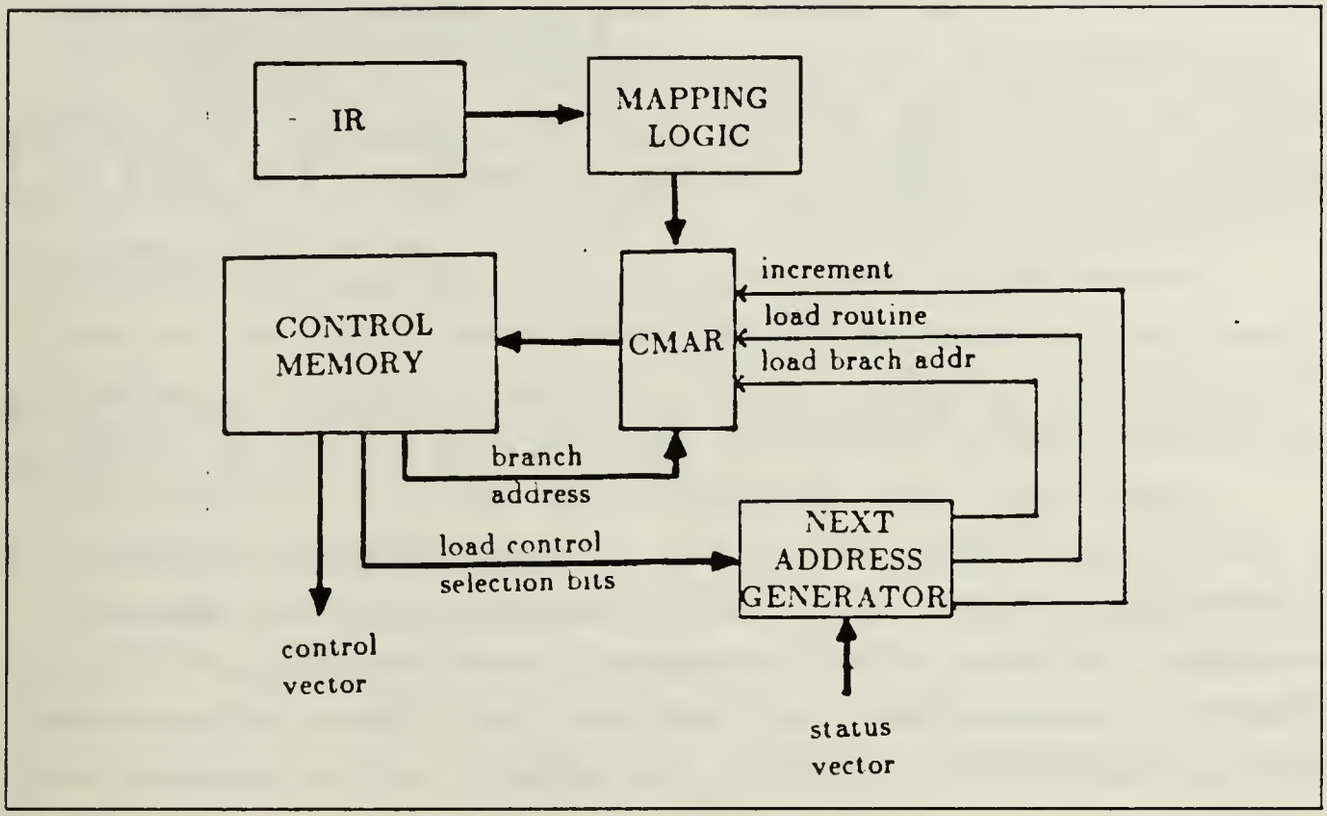


Figure 2.14 A Microprogrammed Control Unit Block Diagram.

among memory devices. While the information is being transfered, it may or may not change. The former case is called a FUNCTIONAL TRANSFER (because some function, logic or arithmetic, of the contents of a source is placed into a destination) and the second case is called a SIMPLE TRANSFER (after this kind of transfer is completed, the destination holds a copy of the source contents). This being the case, a major part of the functional description of a computer will consist of a schedule, or listing, of allowable data transfers under different conditions. It is then convenient to have a symbolic notation to describe these transfers. This section will introduce such a notation.

1. Simple Transfers

Registers are designated by capital letters (sometimes followed by numerals) usually chosen so as to denote the function of the register. Each bit of n-bit register are numbered in sequence from 0 to n-1. Subscripts denote

individual bits of a register. Thus IR_3 means the third bit of the Instruction Register. Portions of a register are refered by specifying, within brackets, the first and last bit. The notation MBR<0:3> refers to the first four bits of the Memory Buffer Register.

Memory words are designated by the name of the RAM followed by the name of the register containing the address within brackets. M[MAR] refers to the contents of memory cell of RAM M whose address is the contents of register MAR.

Buses are also designated by capital letters with the last three always being BUS. The notation to represent the individual lines of a bus is identical to the notation introduced in the last paragraph for registers. Thus INTBUS<3:5> denotes the 3th, 4th and 5th lines of Internal Bus, e.g, bits 3, 4 and 5 of the binary vector carried by the bus.

Functional Units are refered to by their names in capital letters having the prefix FU.

A simple transfer is denoted by an arrow pointing from the source to the sink as shown below:

$$R1 \leftarrow ABUS$$

and parallel transfers, e.g., transfers that are executed in the same control state are separated by commas. For example,

$$CBUS \leftarrow MAR,\ IR \leftarrow R1$$

specifies two transfers that occurs simultaneously. Constants are treated as contents of special registers whose name is the value of the constant. Thus,

$$RB \leftarrow 0$$

denotes the CLEAR operation of register RB.

2. Functional Transfers

The function performed during the transfer is specified within parentheses at the back of the transfer arrow. In the case of a binary operation, the sources are separated by commas. For example:

$$ABUS \leftarrow (+)R1,R2$$

denotes the transfer of the arithmetic sum of the contents of registers R1 and R2 to bus A.

G. TIMING CONSIDERATIONS

In the previous sections, the timing for the individual Data Flow components was presented. When two or more of those devices are interconnected, the timing for the structure necessarily reflects their individuals time constraints.

In the section concerning the Control Unit, it was said that the controllers must stay in a control state the time sufficient for the slowest data transfer to take place. This interval of time is called the DATA CYCLE TIME. The estimation of the data cycle time is the key for determining the system timing.

Consider Figure 2.15, which shows part of the block diagram of an accumulator-based processor, e.g., a processor that has one register, called the accumulator, as the only sink for all functional transfers.

Suppose that the following instruction is intended:

ADD R1

This means that the contents of register R1 is to be added with the contents of the accumulator and the result placed in the accumulator.

To carry out this instruction the following data transfers are necessary:

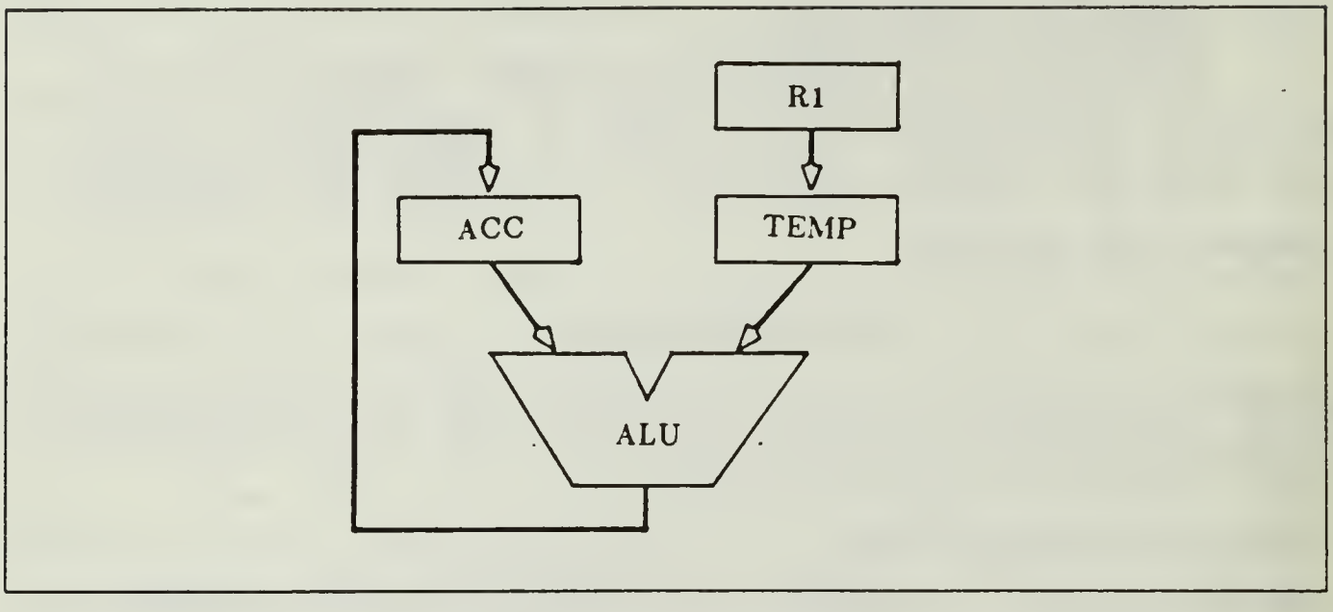


Figure 2.15 Part of a Data Flow.

1) TEMP ← R1

2) ACC ← (+)TEMP,ACC

The timing diagram for the first transfer is shown in Figure 2.16.

The time to process this simple transfer is:

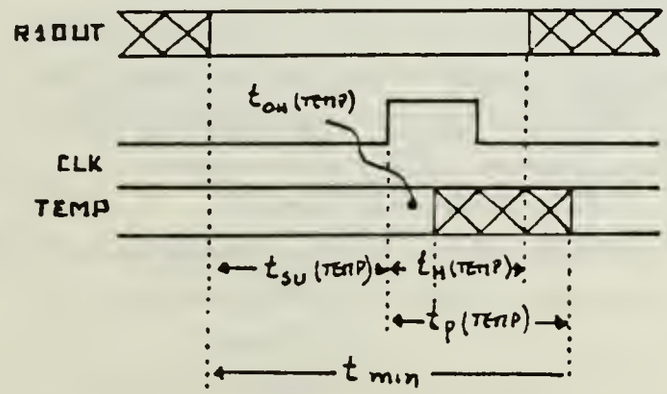
$$t_{smin} = t_{setup} + t_{p(max)}$$

Figure 2.17 shows the timing diagram for the second transfer.

The time necessary for this functional transfer is:

$$t_{fmin} = t_{p(add)} + t_{setup(acc)} + t_{pmax(acc)}$$

The transfer time t_{fmin} is greater than t_{min}, which it is not surprising since in a functional transfer the Functional Unit propagation time must be paid. Because propagations delays are not fixed, it is a safe rule to use worst-case delays. In this example if the addition takes the slowest path within the Functional Unit, then t_{fmin} is the data cycle time for this processor.



$t_{su(TEMP)}$ TEMP setup time

$t_{H(TEMP)}$ TEMP hold time

$t_{on(TEMP)}$ TEMP output hold time

$t_{p(TEMP)}$- TEMP maximum propagation delay

t_{min} - minimum time for transfer

Figure 2.16 Timing Diagram for a Simple Transfer.

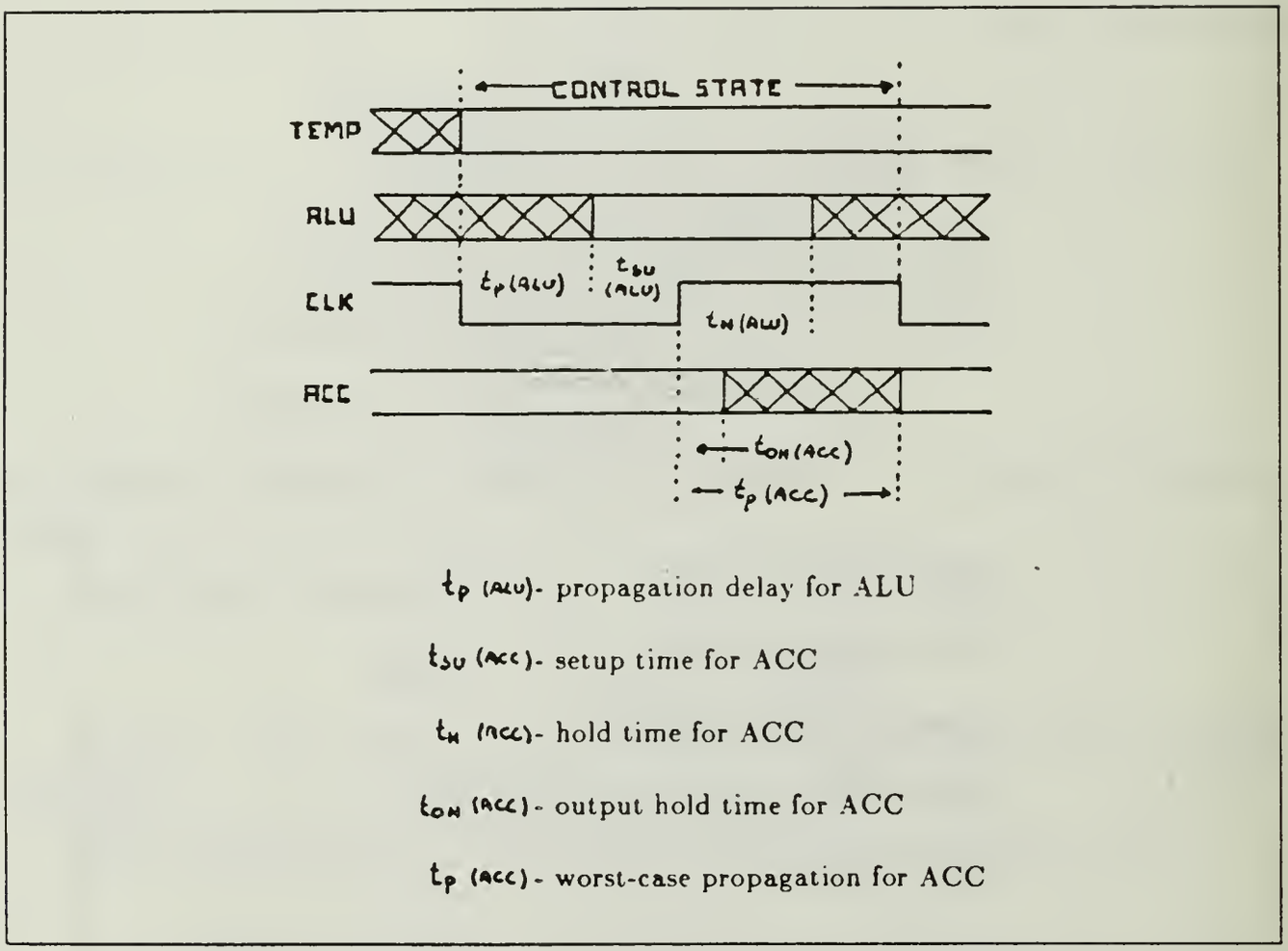


Figure 2.17 Timing Diagram for a Functional Transfer.

Suppose now that TEMP is the source of a bidirectional bus ABUS which have the ALU as one of its sinks. In this case, the propagation delay for the TEMP output gating must also be considered when computing tmin, as shown below:

$$t_{min} = t_{tpg(TEMP)} + t_{p(add)} + t_{setup(ACC)} + t_{p(ACC)}$$

The maximum frequency for the clock to drive the Data Flow is:

$$f < 1/t_{min}$$

and the time to carry out this instruction is equal to $2xt_{min}$ in a single-phased scheme. This instruction time can be reduced using a two-phased clock, e.g., driving ACC and R1 with one phase of the clock and TEMP with the other phase. The timing diagram for this case is as shown in Figure 2.18.

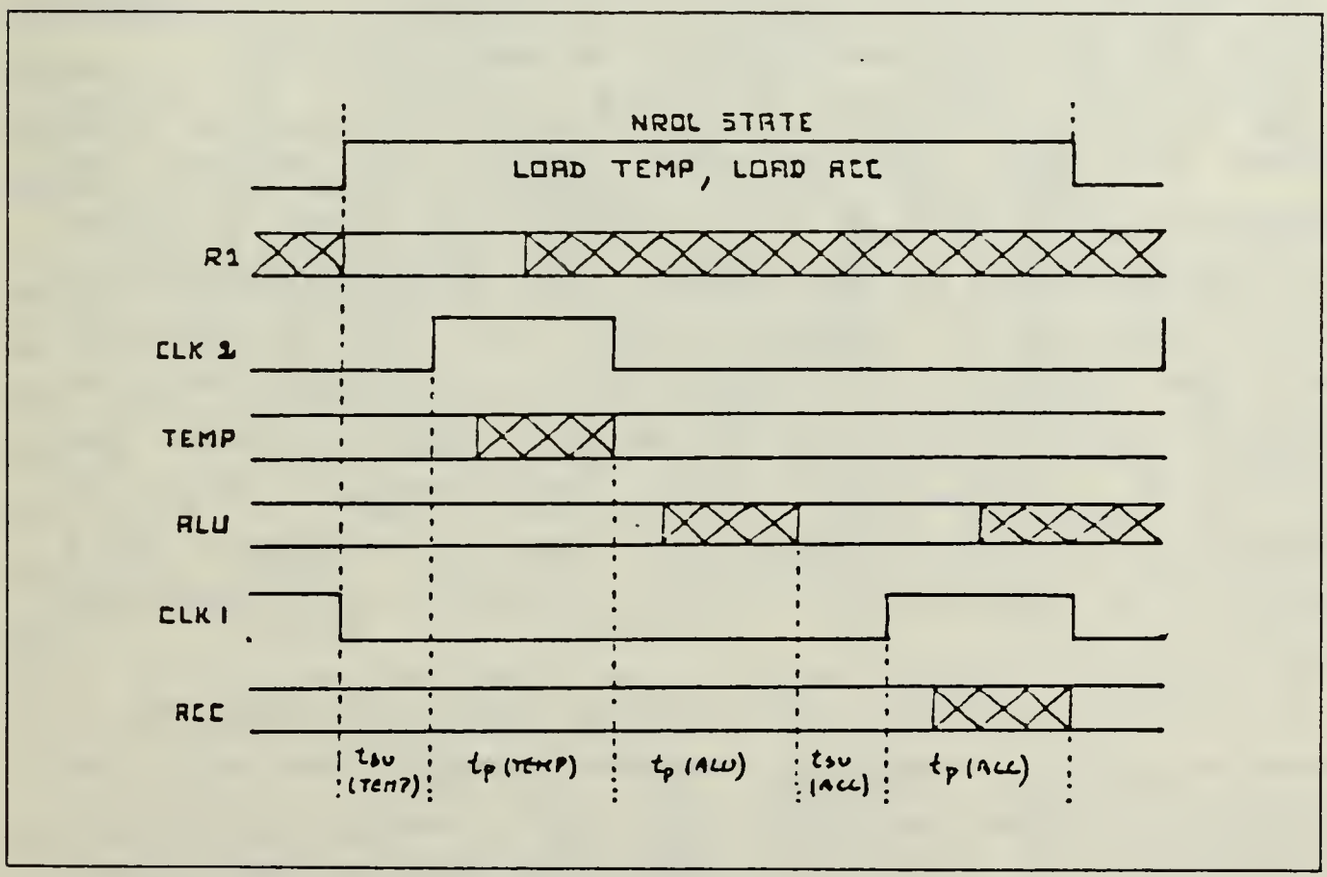


Figure 2.18 Two-phase Clock Timing Diagram.

III. THE DATA FLOW COMPONENTS

A digital system is composed of two distinct structures: the DATA FLOW and the CONTROL FLOW.

A DATA FLOW is composed of two main blocks: the memory block and the functional unit. These blocks are interconnected by buses.

The behavior of the system is characterized by transfers of binary information between memory devices. Typically each of these transfers is enabled by the controller.

Essentially two kinds of information transfer exist:

- from two sources to one sink.
- from one source to one sink.

In the first class, the data is operated on by a binary operator and in the second case by an unary operator.

While the information is being transfered, it can be changed. This means that the memory device that receives the information (the sink of the transfer) is going to store some transformation of the source. This kind of transfer is called a functional transfer and the path through which it took place is called a functional path. A transfer where no change of information takes place is a simple transfer and the path is a simple path. These transformations occur in response to control signals and they are performed by a subset of the combinational logic grouped in the functional unit.

A computer performs its task by means of parallel or successive data transfers between memory devices. The controller selects the paths under user program control. It decodes the user command and sends the appropriate control signals to establish the appropriate paths for completion of the intended command.

This means that it is necessary to choose the paths and the data to be operated upon, or in other words it is necessary to choose which devices can deposit data onto the buses and which can accept the data. To perform this data selection the controller needs to know the topography of the system, which includes a definition of buses and what is attached to them.

A. BUSES

Buses are devices that are used to interconnect the different devices in a digital system. They do not alter data nor remember it; they are essentially wires.

The information in the system is as binary vector and this fact determines one of the characteristics of the buses: the width of the bus, e.g., the maximum length of the binary vector that can flow in it.

Buses serve as interconnections between the devices of the system and therefore the knowledge of which devices are connected by the bus is important: which devices can deposit information in the bus, the SOURCES of the bus, and which can retrieve information from it, the SINKS.

Knowing the bus width, the sinks and the sources is not enough to characterize a particular bus. The way the device is attached to the bus is also important. As mentioned, the data are binary vectors and each of the elements of the vector is a bit. The bits are specified in the vector according to their position and each position has a different weight. Therefore, changing the order of the bits implies changing the information. For a simple transfer, bit 0 of the device should be attached to bit 0 of the bus, bit 1 to bit 1, and so forth.

Figure 3.1 shows two 4-bit registers connected to a bidirectional bus. The data path from R1 to R2 corresponds to a simple transfer because the data flowing through it does not change, in other words, after the transfer R2 holds

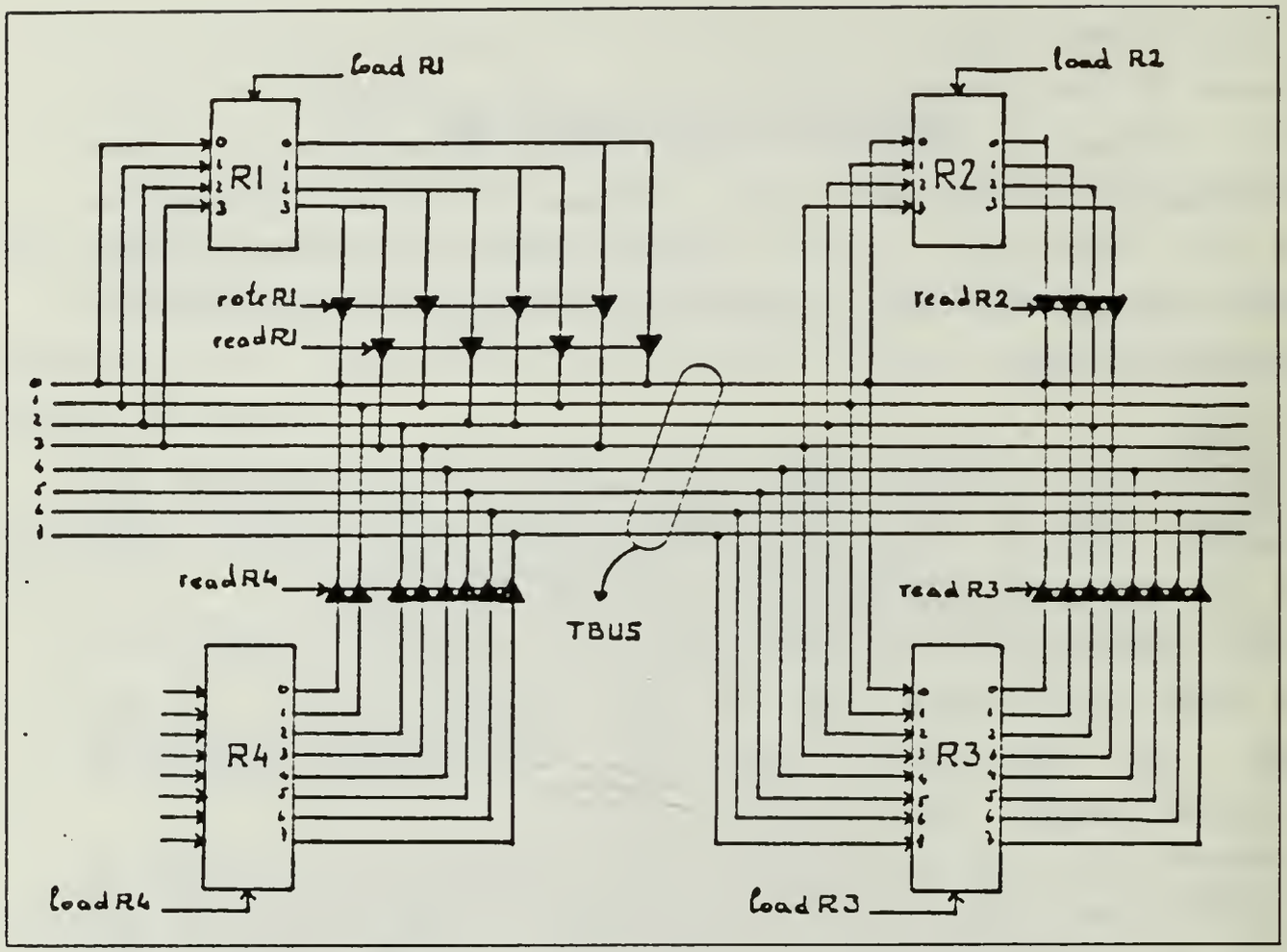


Figure 3.1 Example of Bus Attachments.

a copy of the content of R1. The same is not true for the data path from R2 to R1. This path is a functional one which means that after a tranfer through it, R1 will hold, not a copy of the content of R2 but a transformation of it, in this case the result of rotating its content one bit to the right.

Sometimes, functional transfers are performed by special attachments; this is the case of the barrel shifter presented in Figure 3.2.

As can be seen, with this device the attachments of the buses are not fixed but rather they are changed according to external control signals. This necessitates additional information on the control signals to determine the transfers to/from the bus.

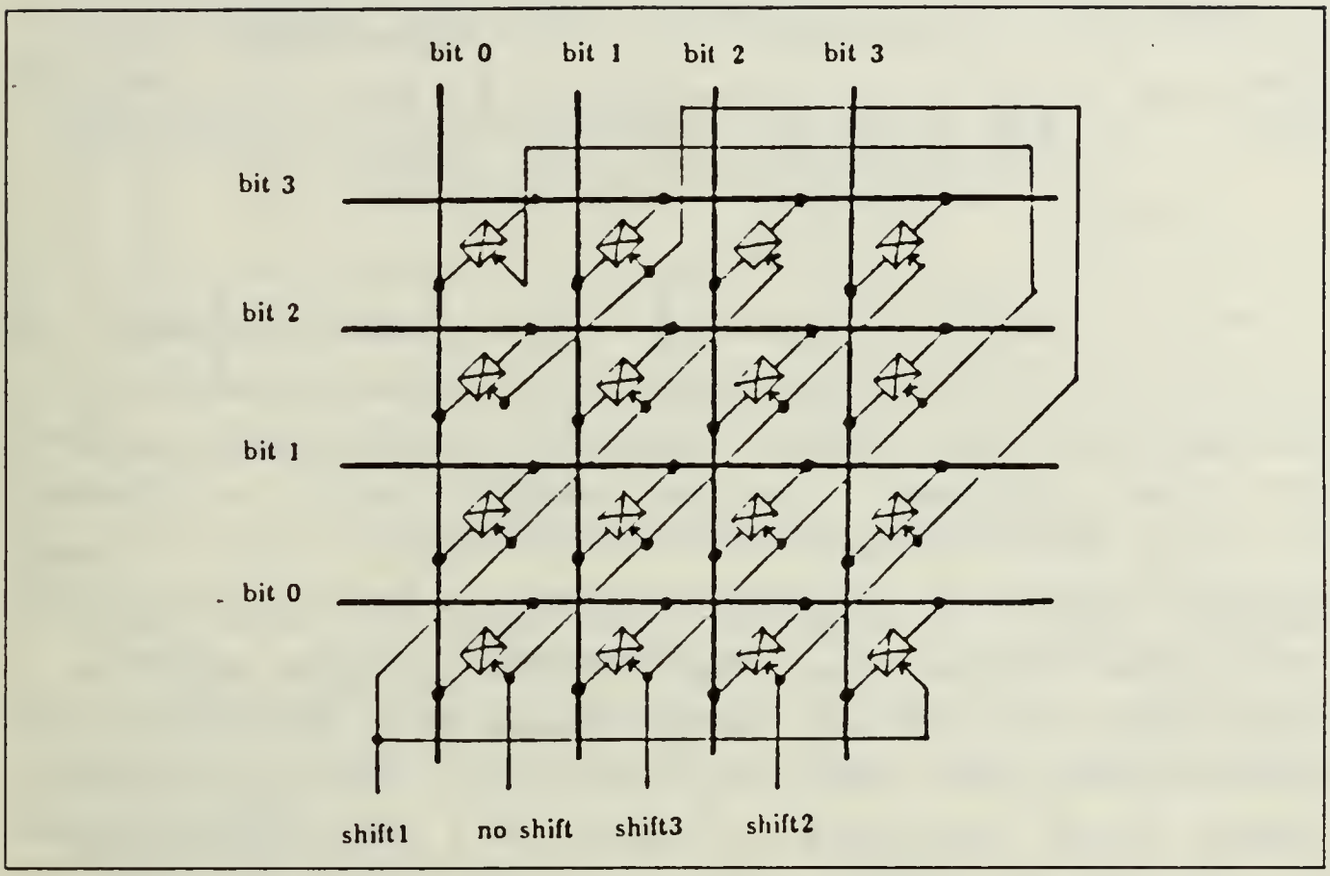


Figure 3.2 The Barrel Shifter.

In summary, a bus is described by:

- name of the bus.
- the sources.
- the sinks.
- how the sinks and sources are attached to it.
- the control signals that allow the transfer to/from the bus.

The syntax to represent this information is shown in Figure 3.3.

The boxes represent fields that are further defined, the circles show the separating characters and the ovoids represent words. To be consistent with the notation introduced in the last chapter, memory devices are designated by capital letters or numerals, the buses are also designated

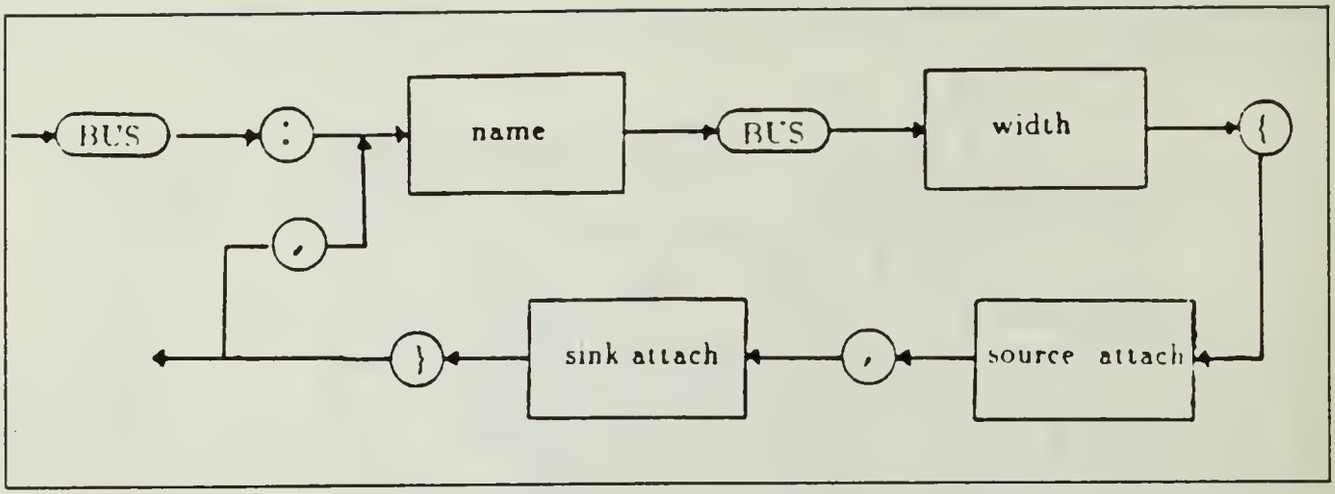


Figure 3.3 The Syntax for Bus.

by capital letters or numerals with the last three being always BUS and the functional units follow the same rule but with the first two letters being always FU. All devices that are sources for the bus are listed in the source attachment field, separated by commas. The sink attachment field lists the bus sinks and has the same syntax as source attachment. Appendix A shows the syntax for each of the boxes of Figure 3.3. Each sink or source attachment begins by specifying the actual portion of the device that is connected, followed by a list of the sources/sinks that are connected to that specific portion. Each of these lists specifies in turn which bits take part in the attachment and the explicit control signals that determine the transfers to/from the device.

In order to simplify the language, default values are introduced by omitting the respective field. If no control is specified, then the "read" signal is implied for sources and the "load" signal is implied for sinks. Implicit and explicit signals will be discussed in detail in Chapter 5. The absence of a subvector means the whole vector flows through the connection; if DBUS has width 8, then DBUS<0:7> can be described just as DBUS.

For example, the bus in Figure 3.1 is described as follows:

```
BUS: TBUS<0:7>{[<0:3>(R1<3:0>•(rotrR1),R1,R2),
              (R3,R4)],[<0:3>(R1,R2),(R3)]}
```

B. MEMORY DEVICES

The memory devices are semiconductors devices capable of storing information. Each memory device must be identified through an unique address and has a specific length (the length of the binary vector it can remember). The syntax for each name, follows the rules introduced in the last chapter.

Memory can be represented as a two dimensional array. One dimensional arrays corresponds to registers and the two dimensional arrays corresponds to RAMs, ROMs and similarly organized devices.

This differentiation is not sufficient because two dimensional memory can be the LIFO (last in first out memory), the FIFO (first in first out memory), the random-access memory (RAM) and the read-only memory (ROM). Other type of memories such as Content Addressable Memories may also occur in some applications.

1. Registers

Registers can be viewed as boxes capable of storing one binary vector and having one input data vector, one output data vector and three control signals: the clock, the LOAD signal and the READ signal. Registers are specified by:

- The name of the register.
- The length of the word it can hold.
- The phase of the clock driving it.
- The type of register, e.g., falling-edge or rising-edge triggered.
- The sources.
- The sinks.

It is assumed that the input data vector is stored by the edge of the clock when the LOAD signal is activated and that the register contents can be sensed by the "outside world" while the READ signal is active.

The syntax to describe a register is shown in Figure 3.4. The "clock" field specifies the phase of the clock driving the register and which edge of the clock triggers it, as illustrated in Appendix A; an "r" in the subfield "edge" denotes a rising-edge triggered register and an "f" stands for falling-edge. If a single-phase scheme is in use all registers receive the same phase of the clock. This phase is designated by the digit 0.

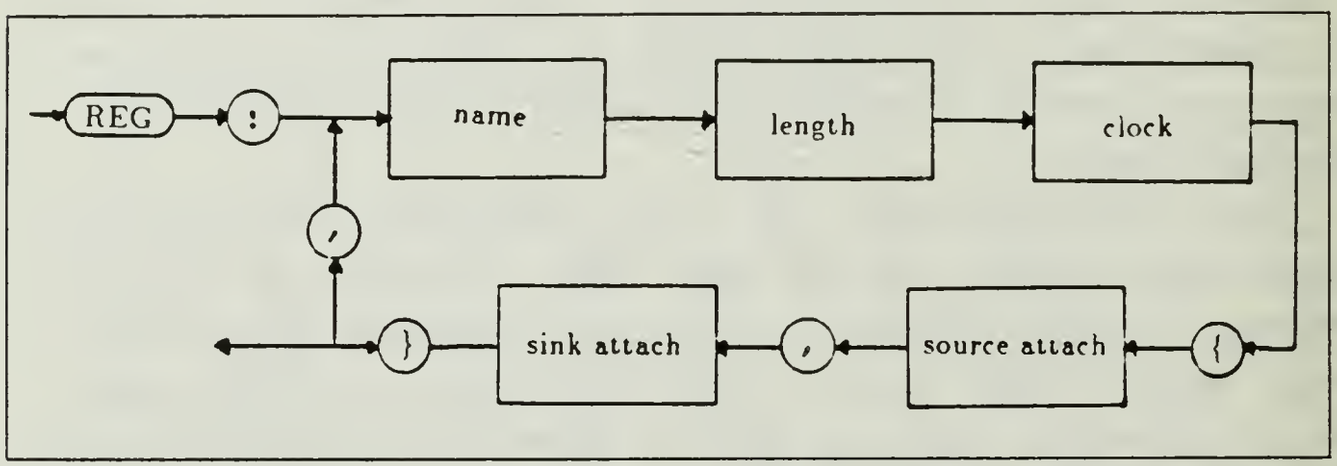


Figure 3.4 The Syntax for Register.

As an example, R1 in Figure 3.1 is described as below (suppose that R1 is rising-edge triggered by phase 1 of the clock):

```
REG: R1<0:3>(1,r){[<3:0>(TBUS<0:3>•(rotrR1),TBUS<0:3>)],
                  [(TBUS<0:3>)]}
```

2. LIFO's

The LIFO memory, also called stack memory, is an array of registers where the information flows in two directions, under the control of two signals, the PUSH and the POP signals, as illustrated in Figure 3.5.

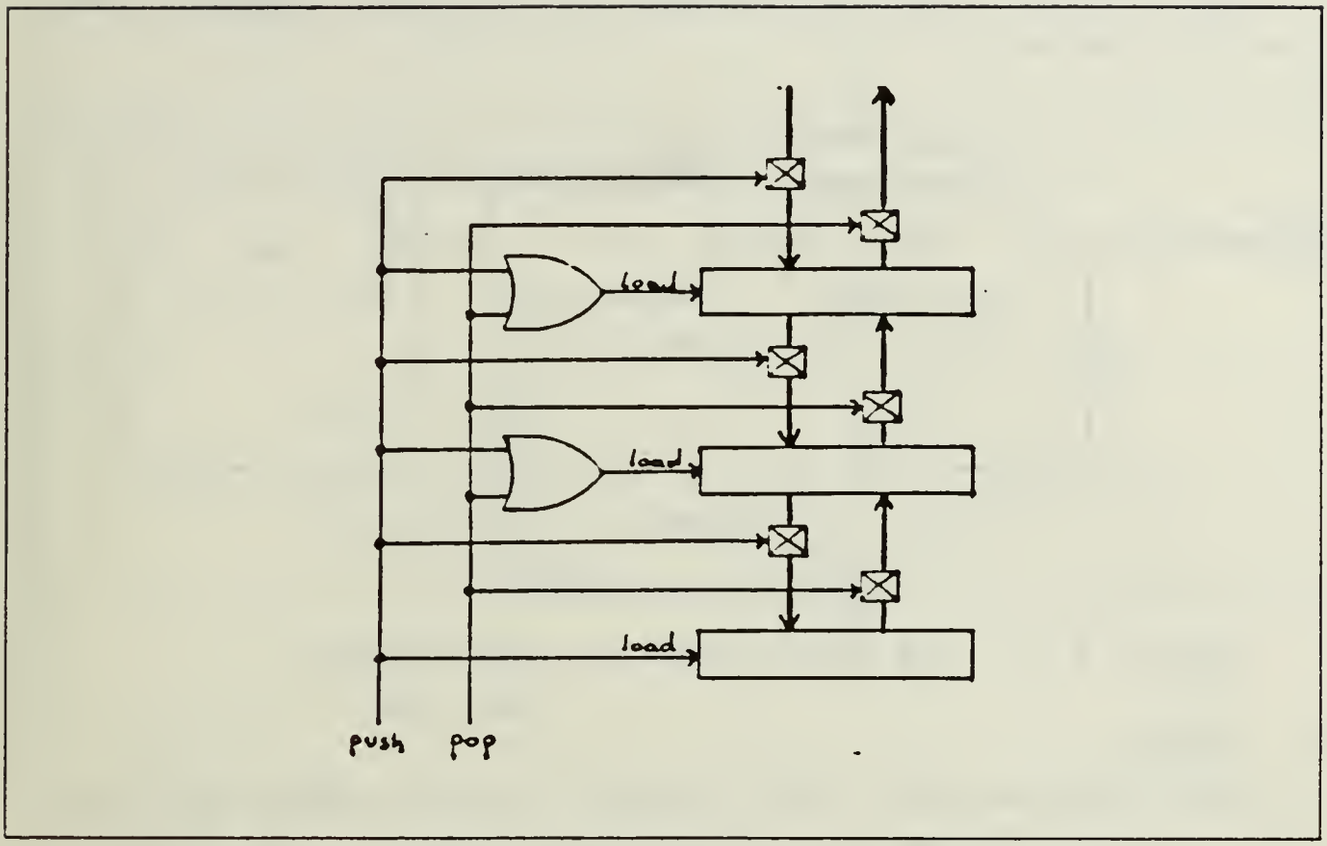


Figure 3.5 The LIFO Memory.

Input data is written into and read from the same register. Additionnaly the structure may provide a status flag signalling that the stack is full or not full. This structure is identified by a unique name, the address where to route its control signals. The LIFO is specified by:

- the address of the stack.
- length of the word it can hold.
- length of the stack (number of words it can store).
- the phase of the clock driving it.
- the type of registers, e.g., rising-edge or falling-edge triggered.
- the sources.
- the sinks.
- if status flag is provided.

The syntax that describes the LIFO memory is depicted in Figure 3.6.

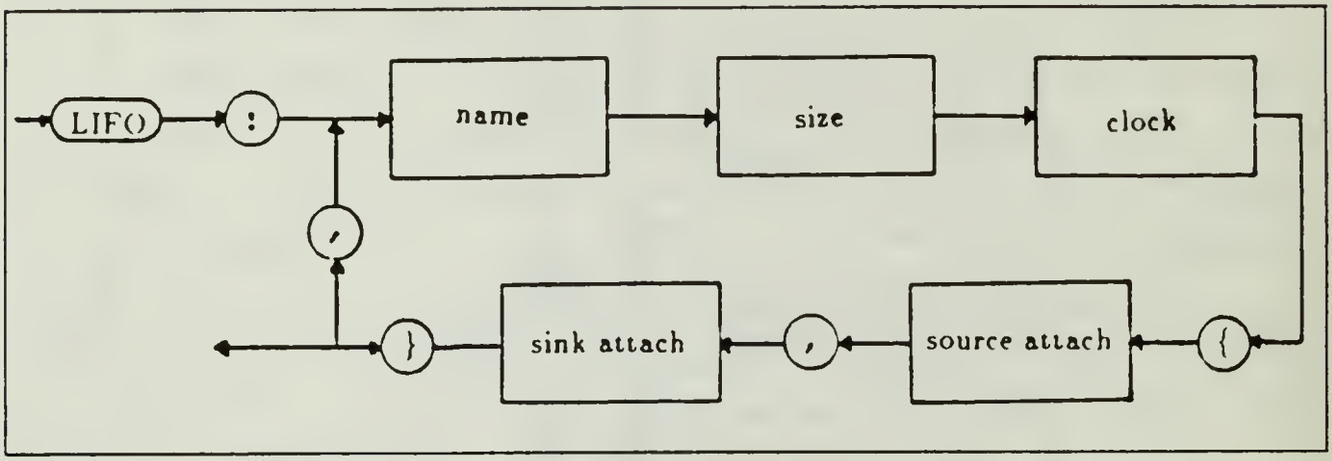


Figure 3.6 The Syntax for the LIFO Memory.

3. FIFO's

The FIFO memory, also called queue memory is an array of registers where the information flows in one direction only under the control of two signals, the LOAD and READ signals as shown in Figure 3.7.
As the stack, it may supply a status information, e.g., queue full or not full. Each queue also have a unique name. They are specified by:

- the address of the queue.
- length of the word it can hold.
- the length of the queue (number of words it can store).
- the phase of the clock driving it.
- the type of registers used, e.g., rising-edge or falling-edge triggered.
- the sources.
- the sinks.
- status information supplied.

The syntax to describe a FIFO is shown in Figure 3.8. Because a stack is a two dimensional memory device, it

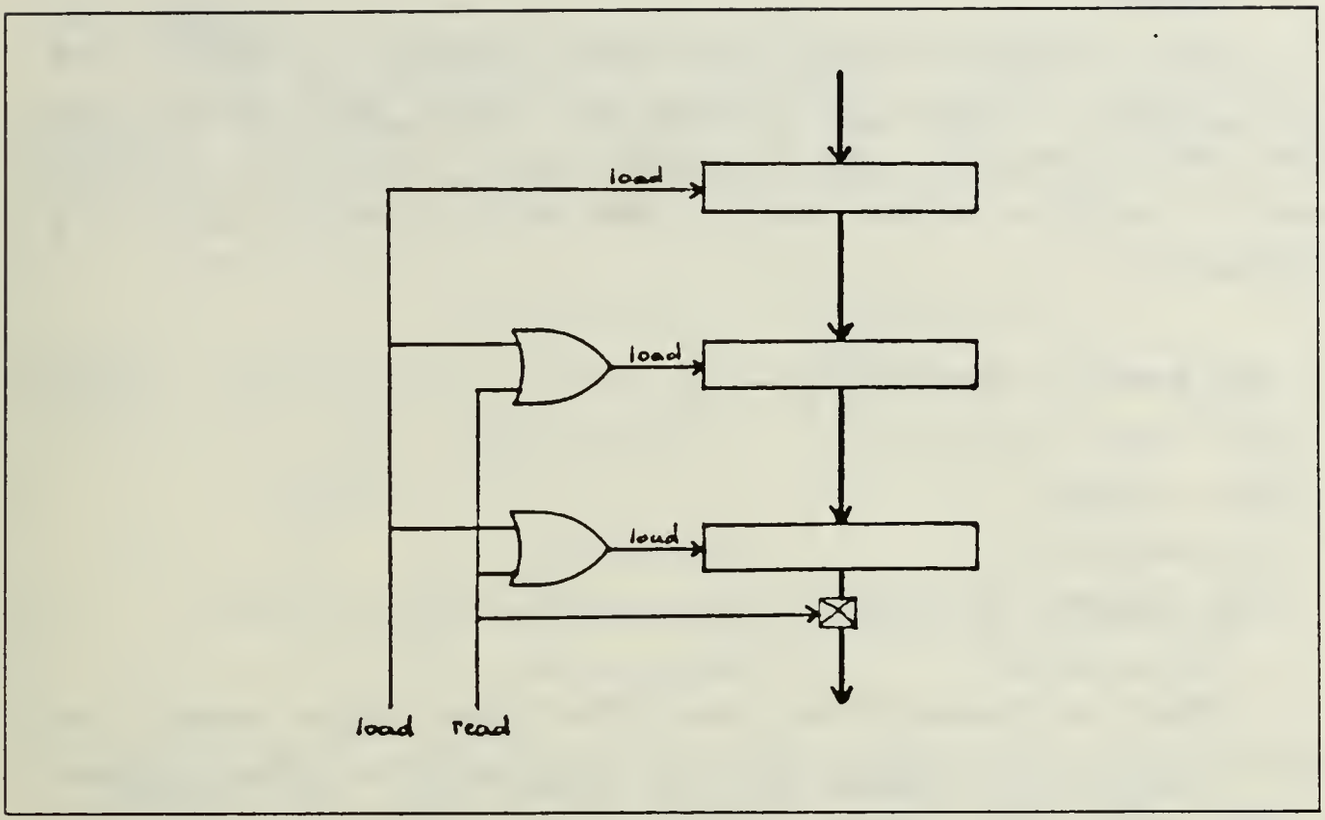


Figure 3.7 The FIFO Memory.

is necessary to specify its size, in other words the number of words it is capable of storing and the length of the words.

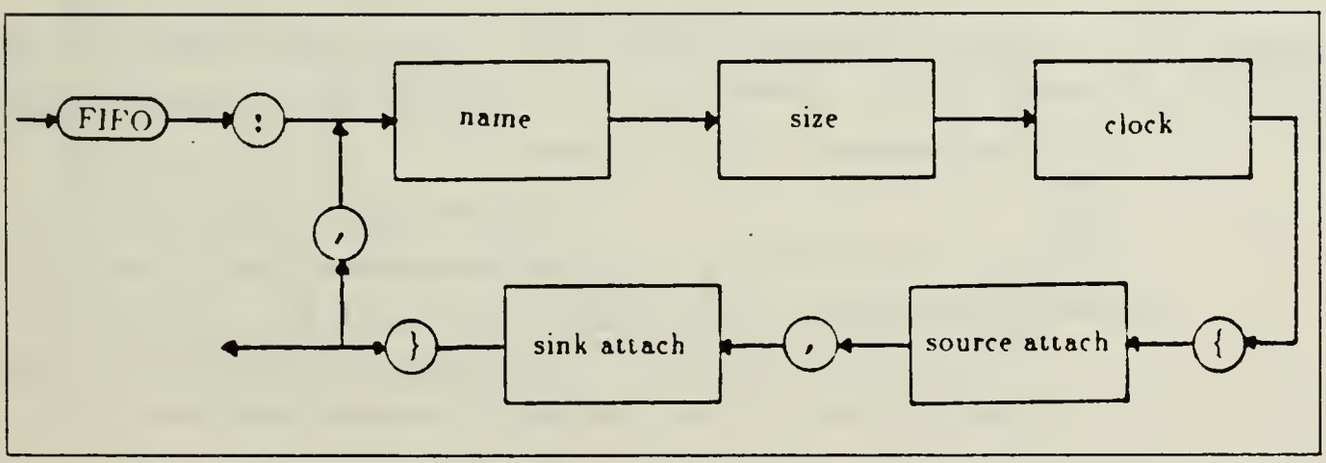


Figure 3.8 The Syntax for the FIFO Memory.

4. RAM's

RAM's can be thought as boxes capable of remembering several binary vectors and having an input data vector, an output data vector, an address vector and two control signals, the Chip Select and the the Write Enable. RAm's are specified by:

- the name of the RAM.
- the size.
- the sources.
- the sinks.
- the address sources.

What is assumed is that the input data vector is stored in the cell whose address is present at the address bus when both the control signals are active and that the contents of the cell specified by the vector in the address bus can be sensed by the "outside world" when the chip select is activated and the write enable is inactive.

The syntax to describe the RAM is in Figure 3.9.

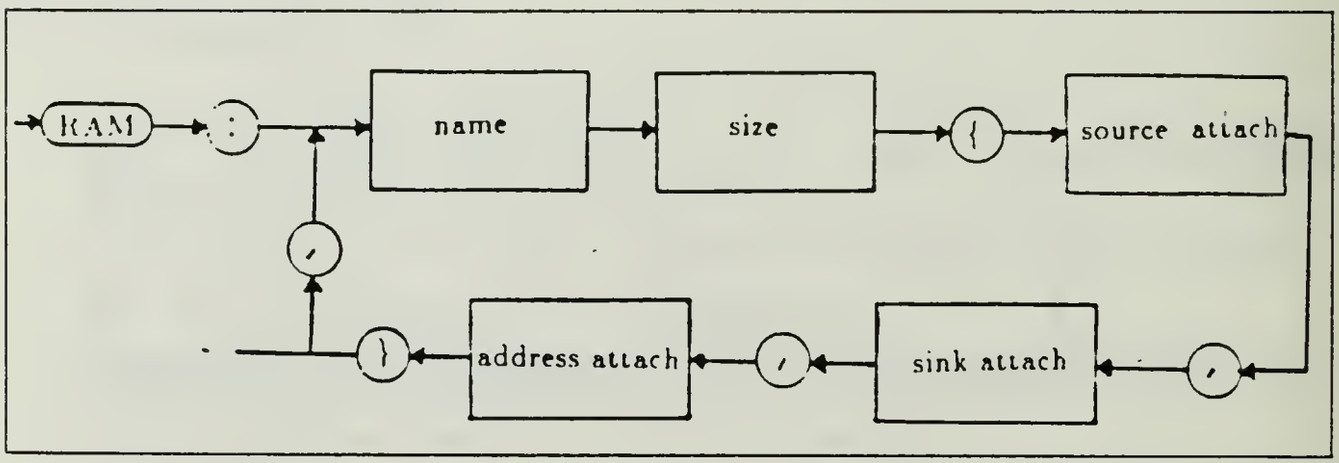


Figure 3.9 The Syntax for the Read-and-write Memory.

The address field lists the devices that are attached to the RAM address input and its syntax is in Appendix A.

5. ROM's

A ROM can be viewed as a RAM with no input facility therefore its syntax is similar to the one introduced in Subsection 4 with no source field as shown in Figure 3.10.

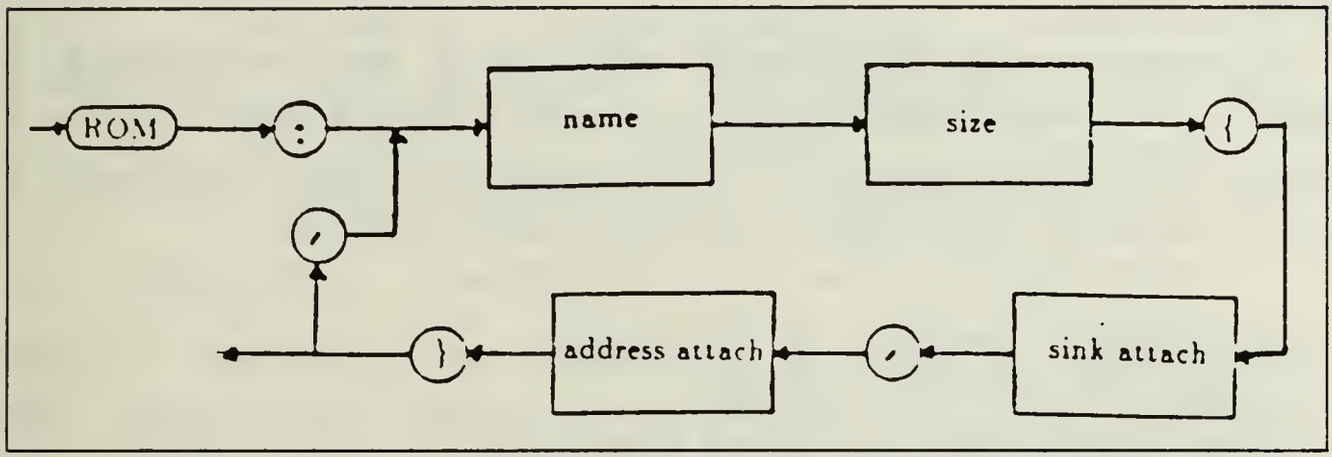


Figure 3.10 The Syntax for the Read-only Memory.

C. FUNCTIONAL UNITS

The functional unit is a combinational logic device that performs logic or/and arithmetic operations on the data flowing through it.

It can be viewed as a box, receiving three input binary vectors and supplying two output binary vectors. The input vectors are:

- two input data vectors.
- one input control vector.

and the output vectors are:

- one output data vector.
- one status vector.

All registers transfers that imply change of information other than those performed by special bus attachament as

described in Section 1, necessarily pass through this unit. Internally, it can be viewed as several possible paths, for the data to move from input A and input B to output S. Each of the paths available corresponds to a different operation and it is chosen by the controller by issuing an appropriate control vector.

In summary the functional unit is a box containing several functional paths between buses A and B and bus S. From the point of view of the controller, the only information it needs to know about the functional unit is:

- operations available on the unit.
- length of the data vectors.
- what status information is furnished.

What is implied is that each operation takes the contents of buses A and B and deposits the result on bus S.

Because many operations only effect a single input, and binary operations may be non-commutative, it is not sufficient to list the operations on the unit. It is also necessary to specify which are the arguments for the operations. In order to make things easy, it is assumed that non-commutative operations have input B as operand and input A as operator. For unary operations, it is necessary to list which operate on input A and the unary operations that operate on input B.

The status vector need not to be changed by every operation. In some digital systems, only a set of the functional unit operations affect the status vector. In this case, the status information must explicitly specify how the status vector is effected.

The syntax that describes the functional unit is shown in Figure 3.11. The "binop", the "unopA" and the "unopB" fields list the binary operations, the unary operations on

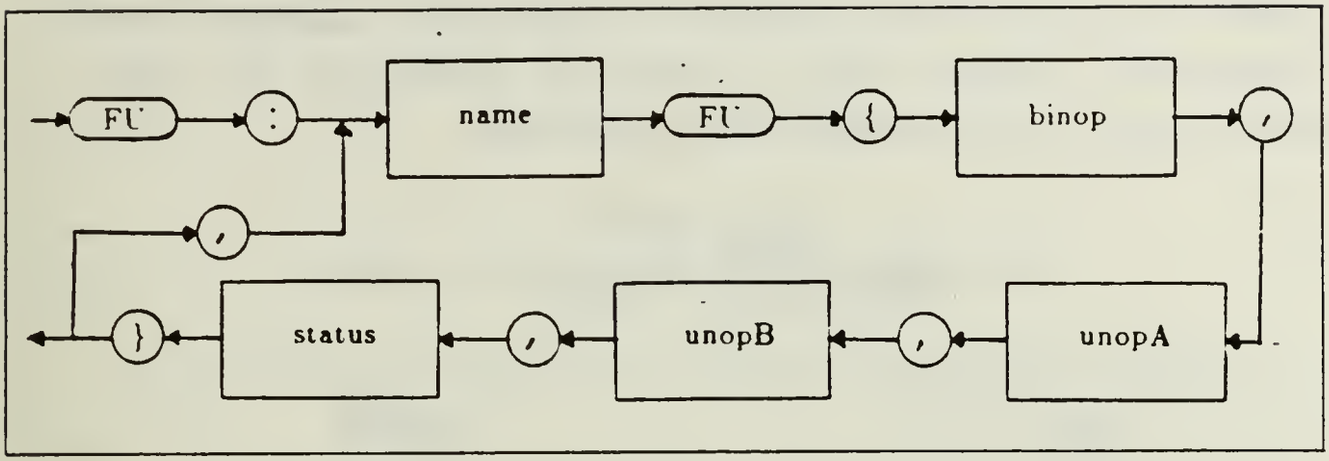


Figure 3.11 The Syntax for the Functional Unit.

input A and the unary operations on input B respectively. The status field shows the information on the functional unit output vector together with the operations that affect it. For example, a functional unit called ALU, performing the arithmetic addition, the AND and the complement operation on both inputs and providing information on zero output for all them and the information on overflow for the addition is described in the following way:

FU: ALU{[(+,∧),( )],[Z,O(+)]}

Note that if no operation is specified with a flag it is assumed that this is effected by all the operations performed by the unit. Sometimes a Functional Unit only performs one type of operations or sometimes it does not provide any status information. Square brackets surrounding a blank specify the absence of a field. For example, the following statement:

FU: INC{[(),(+1),()],[]}

specifies a Functional Unit, called INC, that performs just one unary operation on input A and that provides no status information.

Table II lists some of status information commonly found in real-world systems with a possible symbolism for each. Table III does the same for the operations.

TABLE II
SOME COMMON FLAGS

Name	Syntax
Zero	Z
Overflow	O
Carry out	C
Auxiliary carry	A
Sign	S
Parity	P

TABLE III

COMMON FUNCTIONAL UNIT OPERATIONS

Name	Syntax
Addition	+
Addition with carry	+c
Subtraction	-
Subtraction with borrow	-b
Multiplication	x
Increment	+1
Decrement	-1
Division	÷
Decimal adjust	◊
inclusive OR	∨
exclusive OR	⊕
AND	∧
clear	0
Complement	~
Shift right	→
Shift left	←
Shift right with carry	→c
Shift left with carry	c←
Arithmetic shift right	⇒
Arithmetic shift left	⇐
Swap halfs	⇵

IV. THE DATA FLOW

The Data Flow is one of the two conceptual modules that constitutes a digital system. It is composed of memory devices and functional units interconnected by buses. The last chapter introduced a way to describe each of these components. This chapter will try to show how to describe the structure in which these components are embedded.

A. THE DATA FLOW

A digital system is characterized by a particular architecture. Each architecture is a function of two variables: the Data Flow and the Control Flow. In fact, the Operating System, e.g., "those program modules, within a computer system that govern the control of equipment resources" [Ref. 10 :p. 1], also charcterize the system architecture, but at a higher level than the one treated in the present approach. Thus, a computer system architecture can be changed by changing the Data Flow or the Control Unit or both. Therefore any particular Data Flow is always linked to specific architecture.

Being an interconnection of memory devices and functional units, a Data flow is changed either by changing one or more of its components or by changing the way they are interconnected. Every change of a particular Data Flow gives rise to a new Data Flow. This means that a Data Flow has individuality. The Data Flow name manifests its individuality. A Data Flow is designated by capital letters (sometimes followed by numerals) usually chosen so as to denote the architecture it belongs to.

A Data Flow is characterize by a particular structure of its components. The structure is described by listing all the the components and the way they are interconnected.

Because the Functional Unit is the heart of the system, it was chosen to be the first element of the list. following the Functional Unit come the memory devices in the following order: registers, stacks, queues, RAM's and ROM's. Finally the list of the buses gives the description of the interconnections.

Each of the statements of the list was described in the last chapter. Therefore, the characteristics of each component, which are also important in the characterization of the Data Flow, appear in the Data Flow description.

In summary a Data Flow is specified by:

a) the name of the Data Flow.
b) the Data Flow functional unit.
c) the Data Flow registers.
d) the Data Flow stacks.
e) the Data Flow queues.
f) the Data Flow memory.
g) the Data Flow buses.

under the following rules:

1) Each data flow component always receives data from a bus and delivers its content to a bus.
2) Each Functional Unit has associated with it a bus ABUS for input A, a bus BBUS for input B and SBUS as the output bus. If the Functional Unit only performs unary operations it is assumed that its input is input A.
3) If a memory device can deliver its content to more than one destination (bus, memory device or functional unit) then it has associated with it an unidirectional bus as the output bus.
4) If a memory device receives data from two or more arguments (bus, memory device or functional unit) then the memory device has associated with it a bidirectional bus for which it is the only sink.

The following example, illustrates the way in which one might begin to describe a simple computer.

Suppose a simple 12-bit, single-address, single-phased computer named SM1 whose Data Flow is shown in Figure 4.1 Data is written to or from memory via register MD. It has a 3-bit opcode. The ALU performs the following binary operations:

- addition
- subtraction
- logical AND

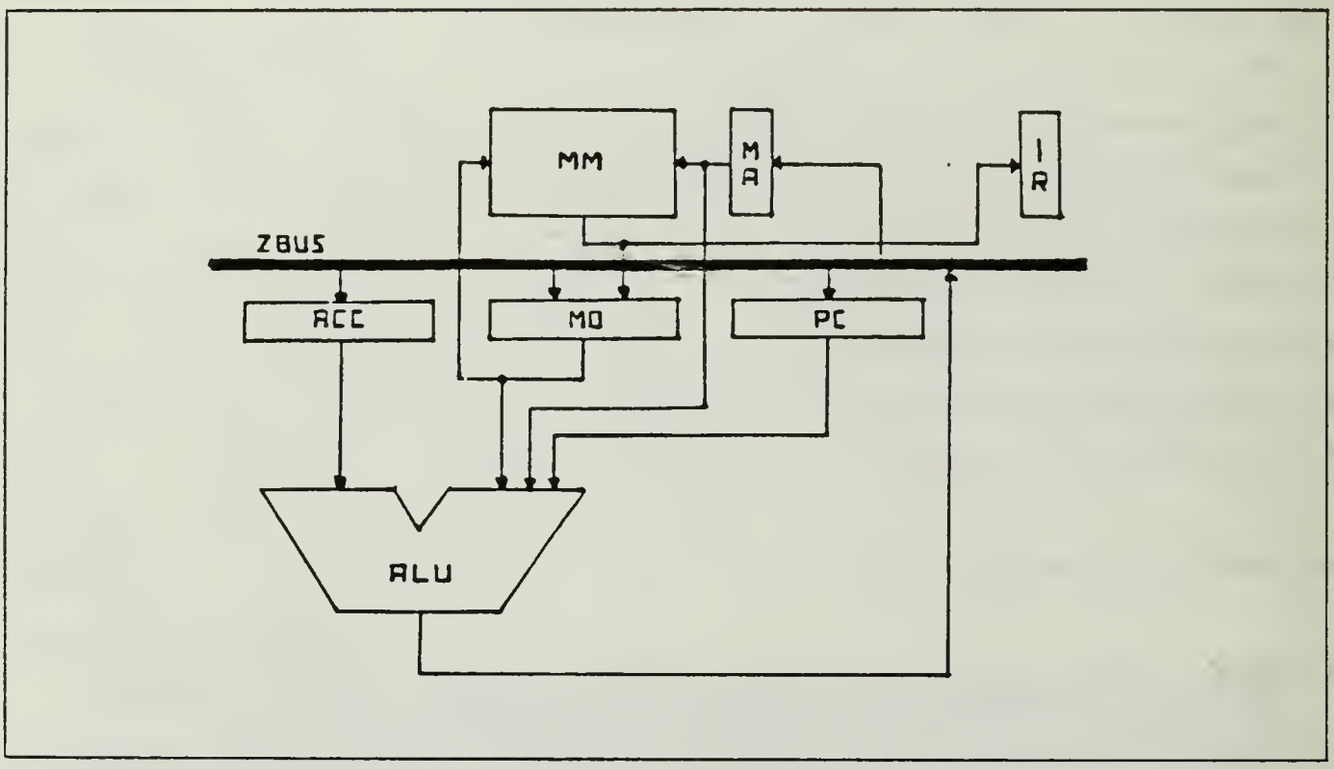


Figure 4.1 The SM1 Data Flow.

the following unary operations on input A:

- complement
- shift right
- shift left
- increment

and the increment operation on input B. It provides three status bits as shown below:

- O (arithmetic overflow) for both arithmetic operations.
- C (carry out of MSB) for both arithmetic operations.
- Z (output zero) for all binary operations and the complement operation.

The program counter PC is incremented by the ALU. All registers are rising-edge triggered. The SM-1 Data Flow is described as follows:

```
DF: SM1;
FU: FUALU{[(+,-,∧),( ,→,←,+1),(+1)],
      [C(+,-),OV(+,-),Z(+,-,∧, )]}
REG: ACC<0:11>(0,r){[(SBUS)],[(ABUS)]},
     MD<0:11>(0,r){[(MDIBUS)],[(MDOBUS)]},
     PC<0:11>(0,r){[(SBUS)],[(BBUS)]},
     MA<0:11>(0,r){[(SBUS)],[(MAOBUS)]},
     IR<0:2>(0,r){[<0:2>(MMOBUS)],[]};
MEM: MM<0:11,0:4096>{[(MDOBUS)],[(MMOBUS)],
                     [(MAOBUS)]};
BUS: ABUS<0:11>{[(ACC)],[(ALU)]},
     BBUS<0:11>{[(MDOBUS,MAOBUS,PC)],[(ALU)]},
     SBUS<0:11>{[(ALU)],[(MA,PC,MDIBUS,ACC)]},
     MDIBUS<0:11>{[(SBUS,MMOBUS)],[(MD)]},
     MDOBUS<0:11>{[(MD)],[(MM,BBUS)]},
     MAOBUS<0:11>{[(MA)],[(MM,BBUS)]},
     MMOBUS<0:11>{[(MM)],[(MDIBUS),<0:2>(IRIBUS)]};
```

B. UNITS

The Data FLow of a digital system can be very complex. The register level description can be more detailed than necessary, sometimes obscuring the intended use of the model. To make things more useful for the designer, another

block is introduced at a higher level of abstraction, the UNIT.

The Unit components are memory devices and functional units interconnected by buses, in an system architecture that can be grouped to form an individual block capable of being operated in parallel (under certain conditions) with other system blocks under the control of the system controller.

The Unit is in turn a Data Flow component as are memory devices and functional units. This means that the Data Flow may include more than one level of abstraction in its description, giving rise to the term MULTI-LEVEL LOGIC.

Functional register is a term by which some authors refer to registers with special features, such as increment and reset capabilities. Typically the program counter (PC) in a digital system falls under this type of registers. In the present discussion a PC register with the mentioned capabilities can be made an UNIT, because it has memory (the register itself) and a functional unit (the combinational logic that performs the reset and increment operations) interconnected by buses. Additionally it can be operated in parallel with other system blocks.

Consider a single-phased computer in which the main memory invariably deposits the content of memory address given by MAR (memory address register) into a register MD (memory data). It has a PC (program counter) capable of being incremented. The fetch cycle of this computer is given by the state diagram of Figure 4.2.

As can be seen, four data transfers take place but only three control states are needed. This example shows that, if an Unit during a particular control state is isolated from the rest of the system, e.g, it is not being used as a source or a sink in a system data transfer, then it can be operated in parallel.

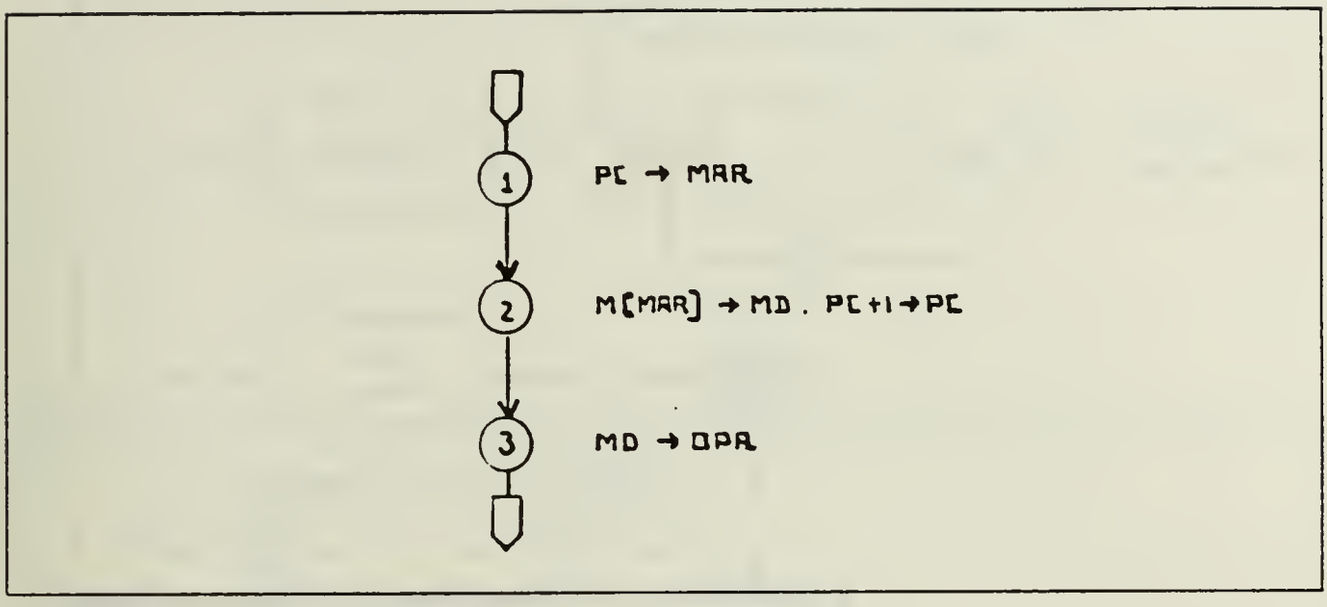


Figure 4.2 Example of a Fetch Cycle State Diagram.

In order to describe a Unit, the following rules need to be formulated:

4) The data always enter the Unit through an INBUS.
5) The data always exit the Unit through an OUTBUS.

These buses need not exist physically: they are used only as abstractions. The following chapter will show how to handle this oddity such that the system description will match the reality.

Because the Unit components are memory devices and functional units interconnected by buses, all the rules and syntax introduced so far apply in the Unit description. The syntax that describes the Data Flow is shown in Figure 4.3.

C. EXAMPLES

The following examples will try to show how to describe real-world systems using the syntax presented so far.

1. The PIC 1650 Microcomputer

The PIC 1650 (Programmable Intelligent Controller) [Ref. 11] is an MOS/LSI microcomputer developed by General

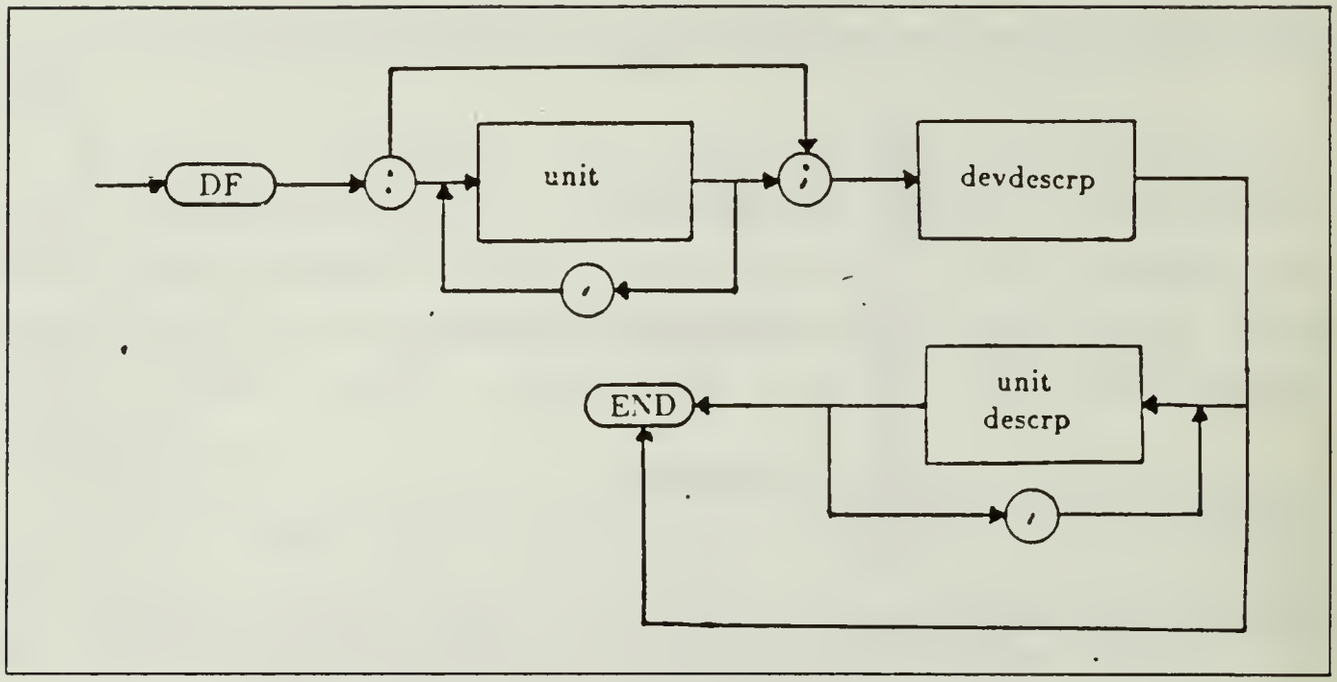


Figure 4.3 The Syntax for Data Flow.

Instrument. It is thought as a good example because its data flow, that is shown in Figure 4.4 , has registers, a read-only memory, a lifo memory and two functional registers as described below:

a. The Program ROM is a 512x12-bit ROM. It is addressed by the Program Counter (F2) and its output vector goes to the Instruction Register(IR).

b. F1 is an 8-bit register with increment and reset capabilities that can be loaded and read under program control.

c. F2 is the Program Counter. It is an 9-bit register with increment capabilities but only the low-order 8 bits can be written to or read from by the program.

d. F3 is the status register. It is an 3-bit register whose bits are modified according to Table IV

e. F4 is an 5-bit register used to generate effective file register addresses under program control.

f. F5-F8 are 8-bit registers used as I/O ports.

g. F9-F31 are 8-bit general purpose registers.

h. W is an 8-bit accumulator.

i. RETST is a LIFO capable of holding two 9-bit words. It is used to store the return addresses.

j. IR is an 12-bit register used as instruction register.

k. The ALU operations are listed in Table IV.

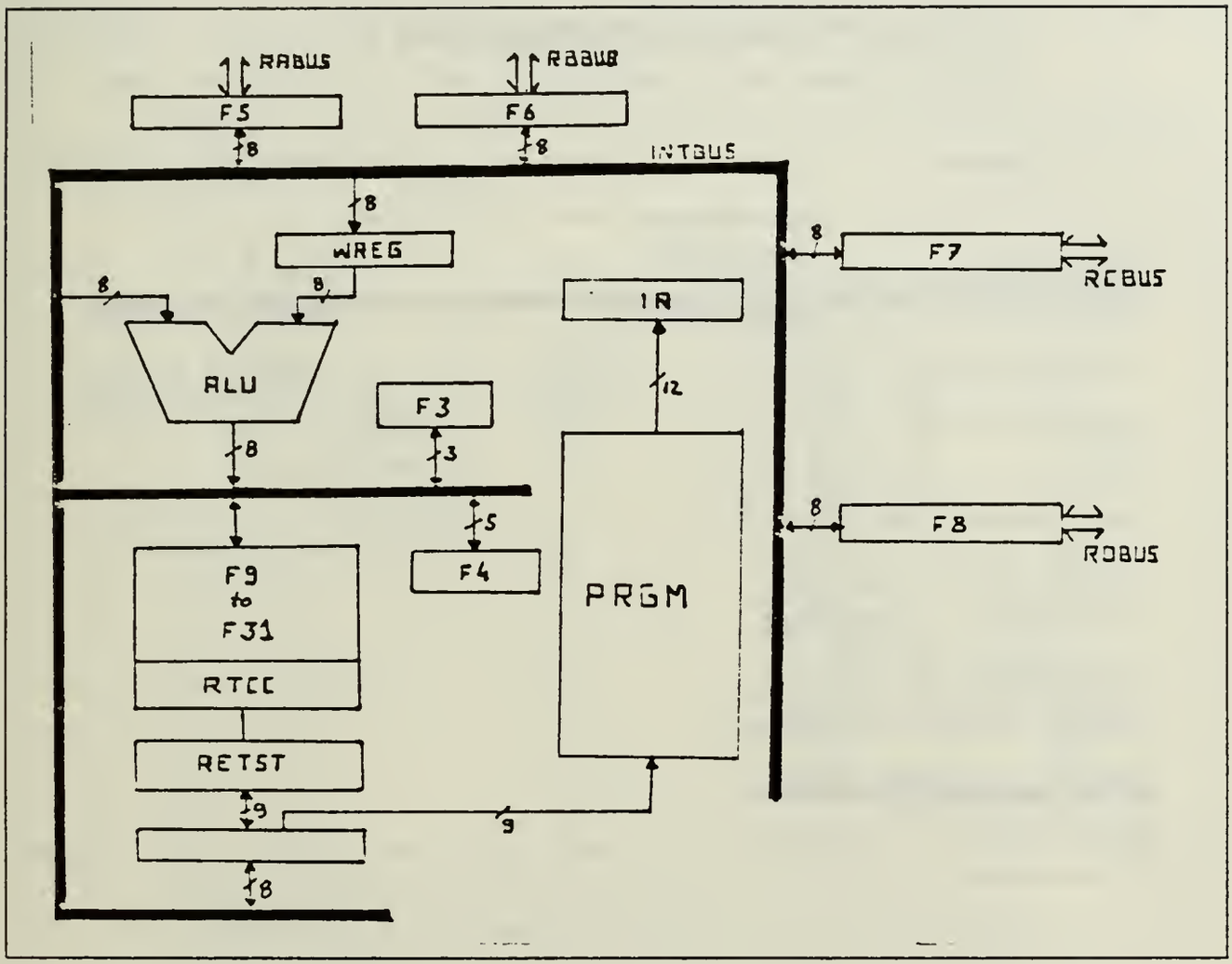


Figure 4.4 The PIC 1650 Data Flow.

TABLE IV

THE PIC 1650 ALU OPERATIONS

Name	Syntax	Status
Addition	+	C,A,Z
Subtraction (W subtractor)	-	C,A,Z
inclusive OR	$\vee$	Z
exclusive OR	$\oplus$	Z
AND	$\wedge$	Z
complement (b input)	$\sim$	Z
clear (both inputs)	0	Z
decrement (b input)	-1	Z
increment (b input)	+1	Z
shift right (b input)	$\rightarrow$	C
shift left (b input)	$\leftarrow$	C
swap halfs (b input)	$\leftrightarrow$	C

It uses a two-phase, non-overlapping clocking scheme. The PC increments on the rising-edge of every phase 1 of the clock while all other registers operate on the rising-edge of phase 2 of the clock. The PIC 1650 data flow description is shown in Appendix C.

2. The INTEL 8085A Micropocessor

The 8085A [Ref. 12] is an 8-bit micropocessor developed by the Intel Corporation to suit a wide range of applications. Its data flow diagram is shown in Figure 4.5 and its components have the following characteristics:

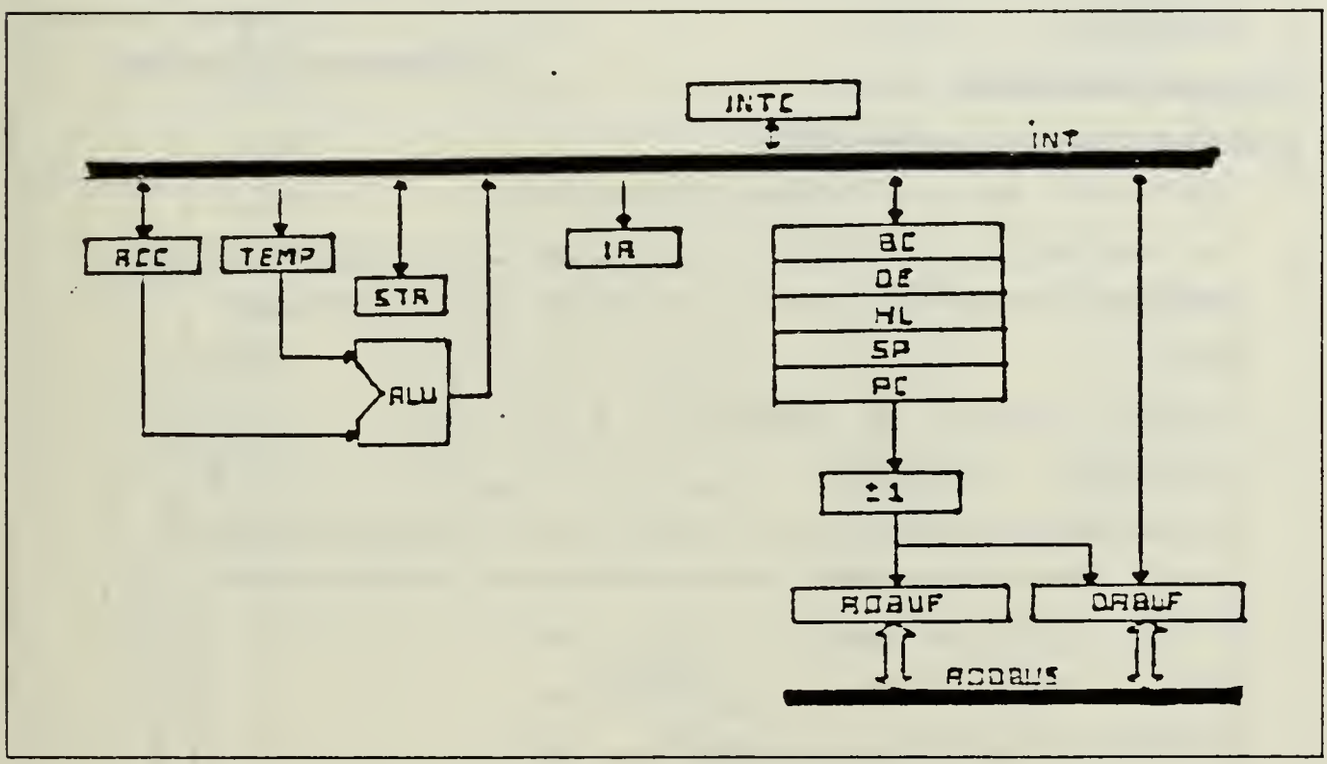


Figure 4.5 The 8085A Data Flow Diagram.

a. PC, SP, WZ, BC, DE and HL are 16-bit registers that can only be loaded 8-bit at a time, and can be incremented by the functional unit INCR.

b. ACC, TEMP and IR are 8-bit registers.

c. STA is the status register. It is an 8-bit register whose bits are modified according to Table V Bits 1, 3 and 5 are not utilized.

d. THe ADBUF is an 8-bit register whose input receives the higher eight bits of the address vector.

e. The DABUF is an 8-bit register whose input receives the lower eight bits of the address vector or the data vector from the INT bus.

f. THe INTC is an 8-bit register holding the interrupt status word.

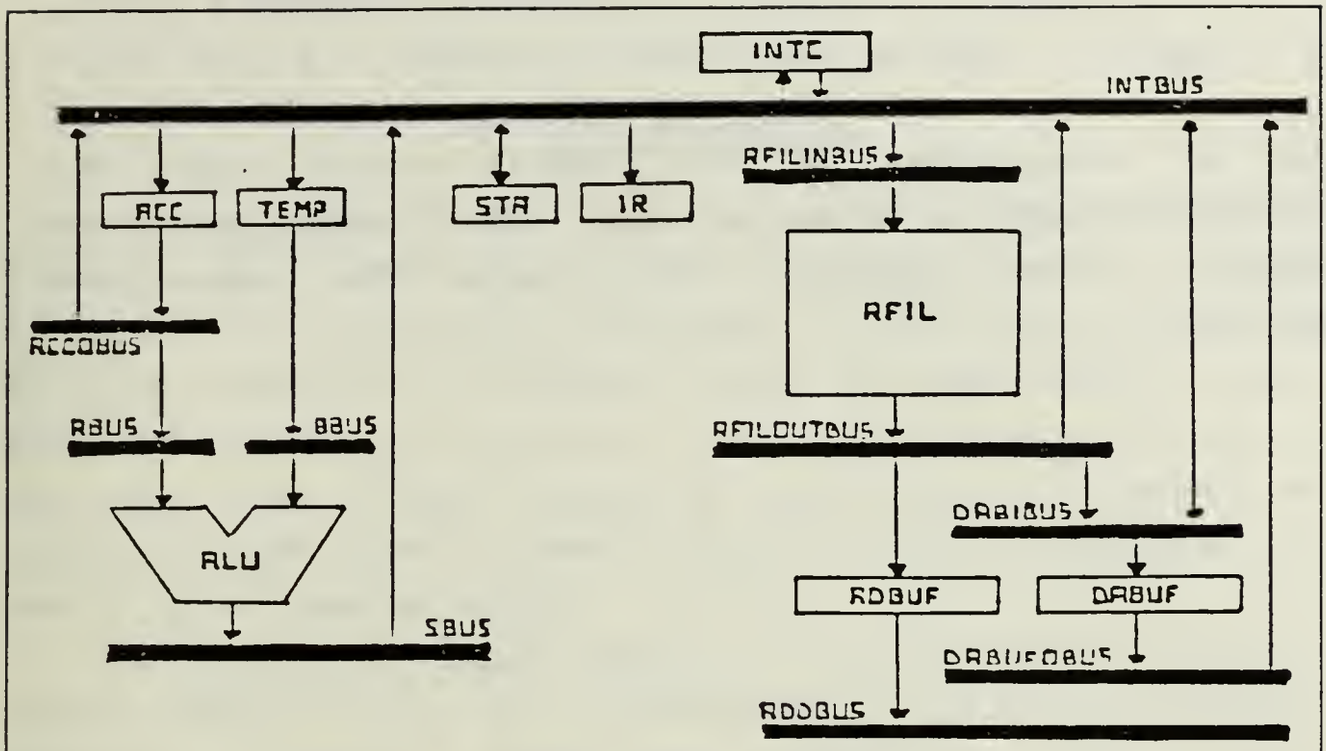
g. The ALU performs the operations depicted in Table V.

TABLE V

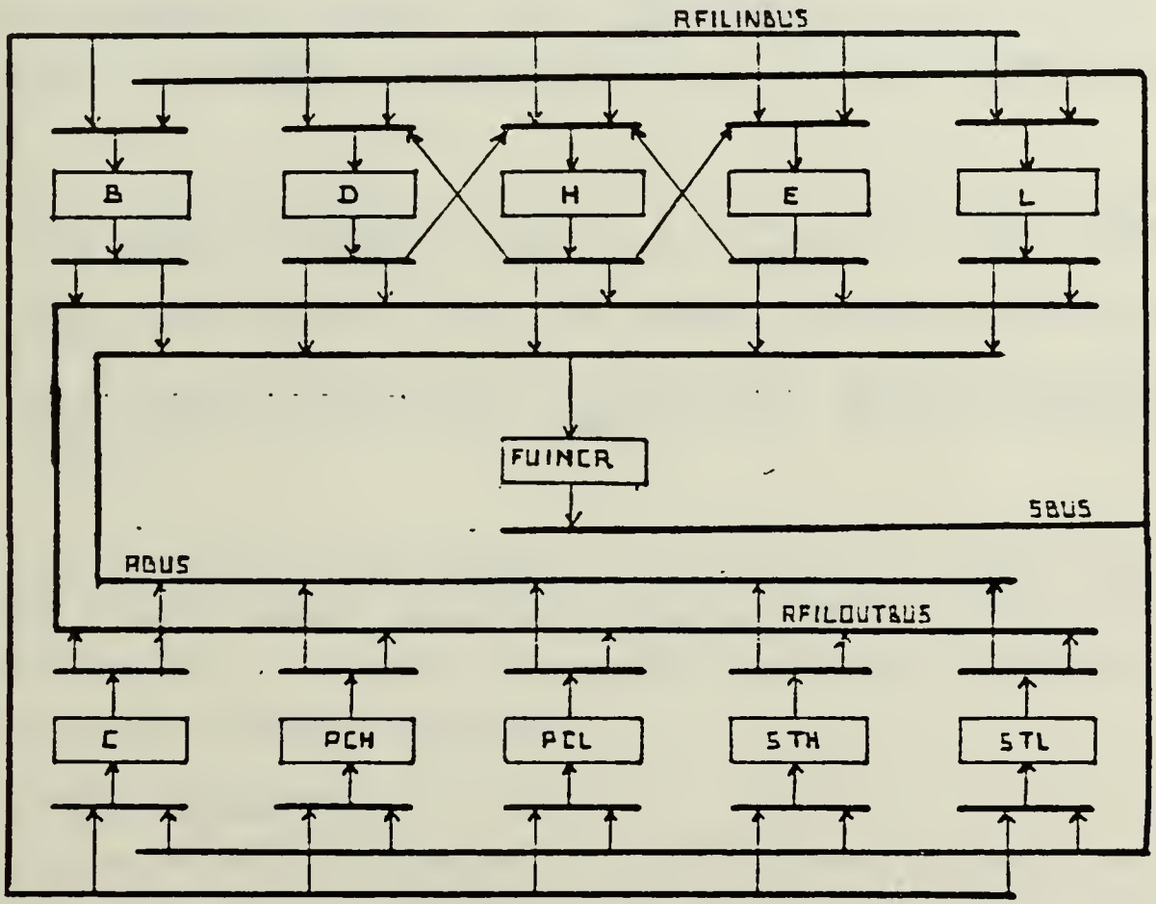
THE 8085A ALU OPERATIONS

Name	Syntax	Status
Addition	+	all
Addition with carry	+c	all
Subtraction (from ACC)	-	all
Subt with borrow (from ACC)	-b	all
inclusive OR	$\vee$	all
exclusive OR	$\oplus$	all
AND	$\wedge$	all
decimal adjust (a input)	$\diamond$	all
decrement (a input)	-1	Z,S,P,A
increment (a input)	+1	Z,S,P,A
complement (a input)	$\sim$	none
shift right (a input)	$\rightarrow$	C
shift left (a input)	$\leftarrow$	C
rot r with carry (a input)	$\rightarrow$C	C
rot l with carry (a input)	C$\leftarrow$	C

All registers are of the falling-edge triggered type and are operated by phase 1 of a two-phase clock scheme, but register TEMP, which receives phase 2 of the clock (see Chapter 5, section F). The description of the 8085A is shown in Appendix B. This description translates into the diagram shown in Figure 4.6.



a) The Data Flow



b) The RFIL Unit

Figure 4.6 The 8085 Descriptive Model.

In Figure 4.6, note the connection between H and D, and L and E. This is not shown in Figure 4.5 but it is necessary because of instruction XCHG (exchange HL with DE) which is accomplished in four control states. Since the fetch cycle comprise three of these, this instruction to be executed in one control state needs the connections mentioned.

V. DATA TRANSFERS

A major part of the description of a digital system consists of a schedule or listing of data transfers. A data transfer is an operation performed in one clock cycle by which the contents of a memory device is taken across a data path and stored in the same or any other memory device in the Data Flow. These transfers consists of a number of transfer steps. One or more control signals corresponds to each of the transfer steps.

The purpose of this chapter is to introduce a way to obtain, from a data flow description, the necessary data transfers, transfer steps and consequently the control signals needed to carried out a particular instruction.

A data transfer is described by a list of transfer steps where:

- The first transfer step of a particular data transfer has always a memory device as a source.
- The last transfer step has always a memory device as a sink.
- All transfer steps of a particular data transfer, but the first and the last ones, do not mention any memory device.

For the sake of simplicity, sections 1 and 2 will consider only inter-register transfers and section 3 will generalize the concepts introduced in these sections for other types of memory devices.

A. SIMPLE TRANSFERS

A simple data transfer is carried out through a data path that does not include any functional unit. To move the contents of R1 to R2 in the block diagram shown in Figure 5.1, the descriptive model gives the following transfer steps:

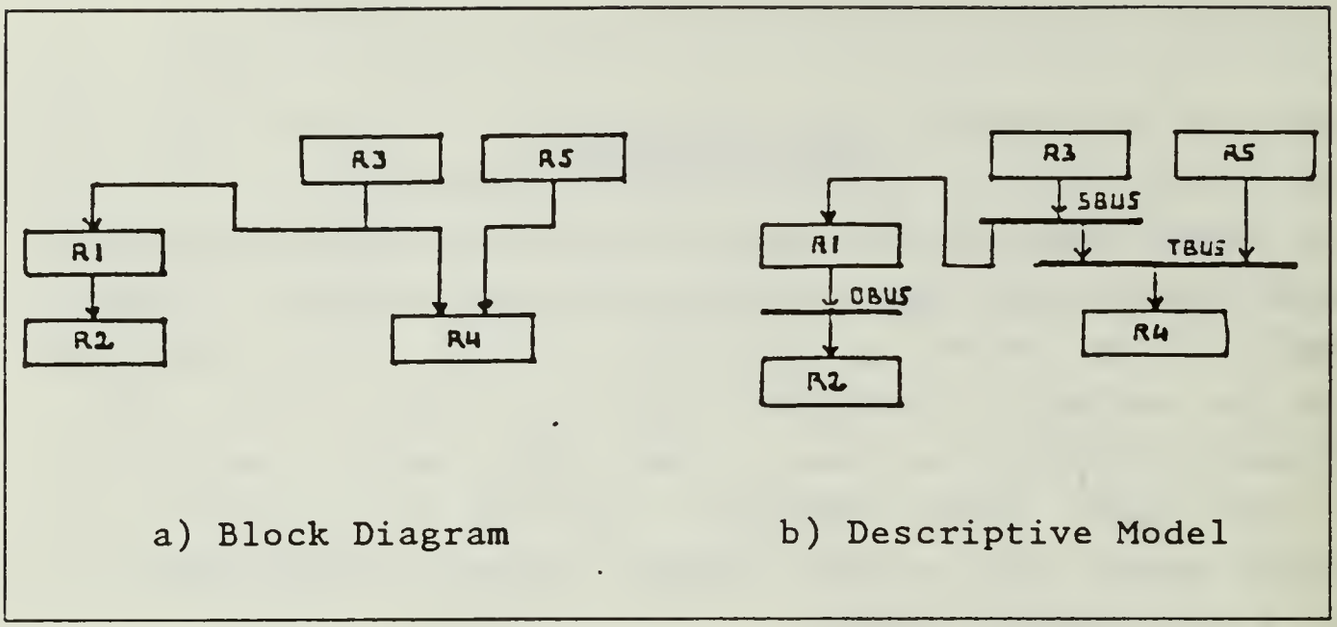


Figure 5.1 Direct Interconnection between Registers.

OBUS ← R1

R2 ← OBUS

Because buses do not have any control inputs, the first transfer step implies the control signal "read R1" and the second one the control signal "load R2". The following rules express this:

(3) Any transfer step between a memory device A and a bus, the device being the source, implies the control signal

read A

(4) A transfer step between a register B and a bus, the register being the sink, implies the control signal

load B

The information about control signals may be incorporated in the symbolic notation introduced in Chapter 2, as shown below:

OBUS ← R1 : read R1

R2 ← OBUS : load R2

The portion of the transfer step statement to the right of the colon is called the control field.

The way registers are connected in Figure 5.1, to move the contents of R1 to R2 needs just one control signal, which is "load R2". The discrepancy between the real implementation and the model is a consequence of rule 1 in Chapter 4. To overcome this disagreement, the following rule is needed:

(2) In a particular data transfer, two transfer steps involving a unidirectional bus B are merged in one single transfer step having as a source the source of B and as a sink the sink of B. The control signals for the resultant transfer step obeys to the following: if the sink is a register then the control signal that is implicit is "load" and if it is a bus then the control signal is "read".

Applying this rule to the example above leads to:

R2 ← R1 : load R2

In the same figure, the transfer steps necessary to move the content of R3 to R4 are:

SBUS ← R3
TBUS ← SBUS
R4 ← TBUS

which simplify to:

TBUS ← R3
R4 ← TBUS

Consider now the following data flow description:

```
DF: EXPL1;
REG: R1<0:3>(0,r){[(R3OBUS<0:3>)],[(R1OBUS)]},
     R2<0:3>(0,r){[(R3OBUS<4:7>)],[(TBUS<0:3>)]},
     R3<0:7>(0,r){[(R3IBUS)],[(R3OBUS)]},
     R4<0:3>(0,r){[(R1OBUS)].[(R3IBUS<0:3>)]},
     R5<0:3>(0,r){[R3OBUS<0:3>)],[(TBUS<0:3>•(rdR5NOP),
```

```
                TBUS<4:7>•(rdR5SWP))]};
BUS: R1OBUS<0:3>{[(R1)],[(R4,TBUS<0:3>)]},
     TBUS<0:7>{[<0:3>(R1OBUS,R5•(rdR5NOP)),
               <4:7>(R5•(rdR5SWP))],[(R3IBUS)]},
     R3IBUS<0:7>[(TBUS),<0:3>(R4)],[(R3)]},
     R3OBUS<0:7>{[(R3)],[<4:7>(R1),<0:3>(R5,R2)]};
END;
```

The model that corresponds to this description is depicted in Figure 5.2.

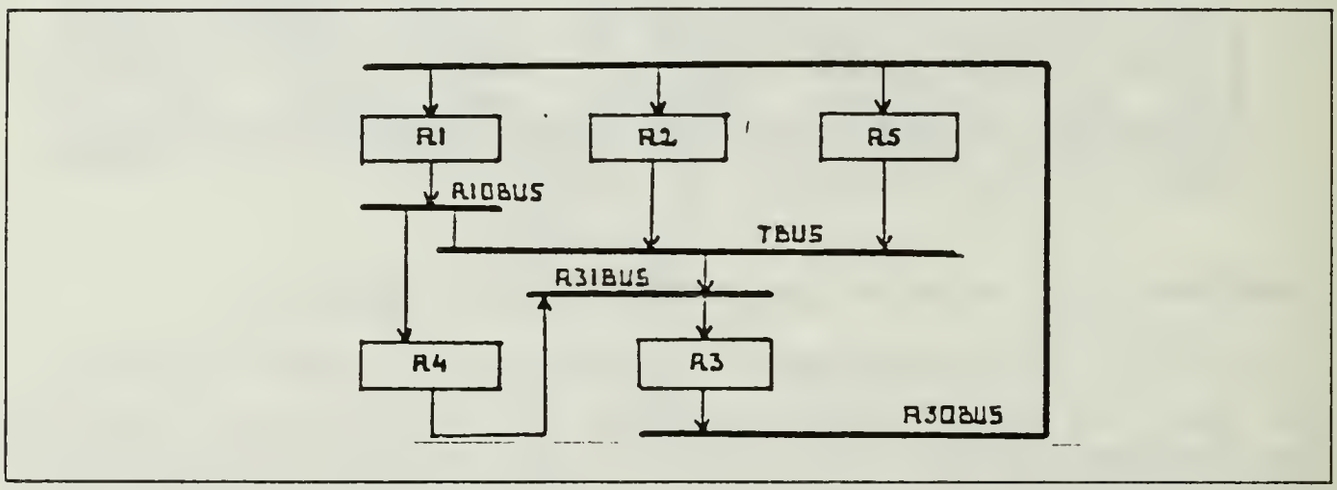


Figure 5.2 EXPL1 Descriptive Model.

To move the content of R1 to R3 the below transfer steps are needed:

$$R1OBUS \leftarrow R1$$
$$TBUS\langle 0:3\rangle \leftarrow R1OBUS$$
$$R3IBUS \leftarrow TBUS\langle 0:3\rangle$$

which after simplification become:

$$TBUS\langle 0:3\rangle \leftarrow R1 \quad :readR1$$
$$R3 \leftarrow TBUS\langle 0:3\rangle \quad :loadR3$$

Suppose now that the contents of R5 is desired to go to the upper part of R3. The following transfer steps are necessary (no simplifications are possible in this case):

```
TBUS<4:7> ← R5     :readR5SWP
R3 ← TBUS          :load R3
```

Note that instead of "readR5", the first transfer step has the control signal "readR5SWP". The reason for this comes from the fact that control signals explicitly specified in the data flow description superpose the implicit ones given by Rules 3, 4 and 5.

B. FUNCTIONAL TRANSFERS

A functional transfer is a data transfer through a data path that includes a functional unit. If the functional unit is a source of a bidirectional bus then it is necessary to provide it with output gating as mentioned in Chapter 2, Section B. This means that the functional unit FU output is available only when the signal "readFU" is active. Because the FU output is of interest only when a specific FU function is chosen, the "readFU" signal is the logical OR of all function-selection signals and therefore it is not a FU external control signal. Figure 5.3 illustrates the above statements.

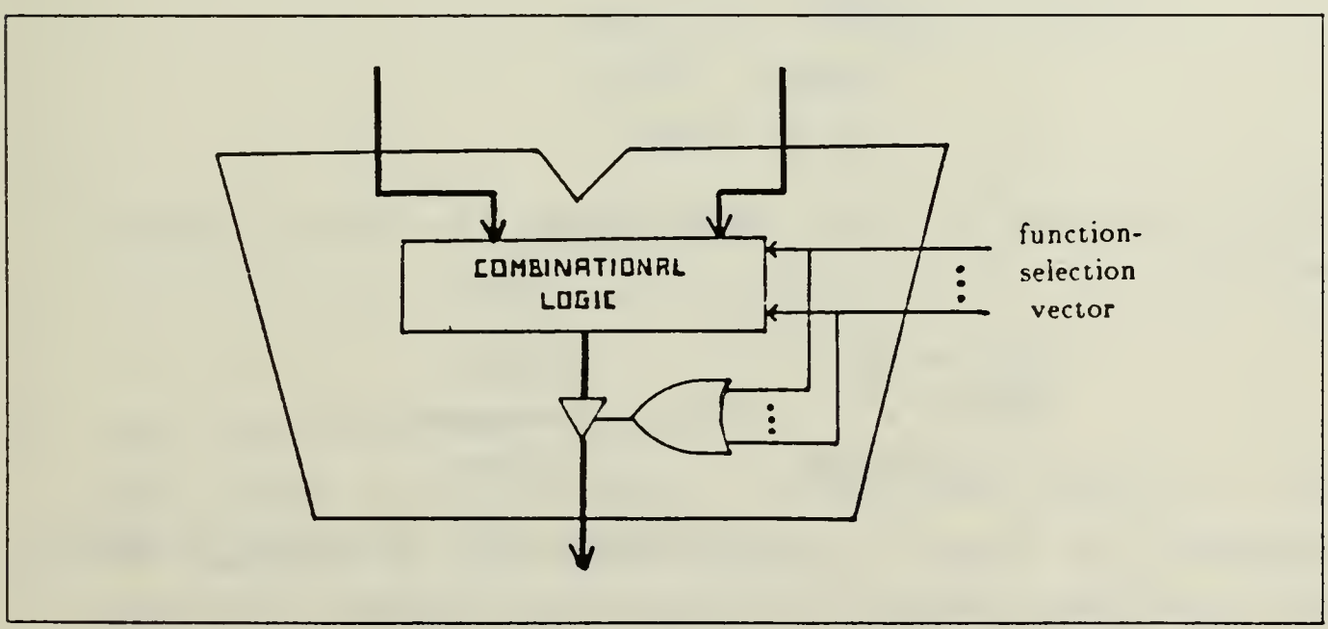


Figure 5.3 Output Gating for a Functional Unit.

In summary the output vector of a functional unit is available only when one of its function-selection signals is active.

Figure 5.4 shows a model with a functional unit called ALU. Suppose that the content of R1 is to be incremented and stored back in R1. This data transfer is accomplished with the following transfer steps:

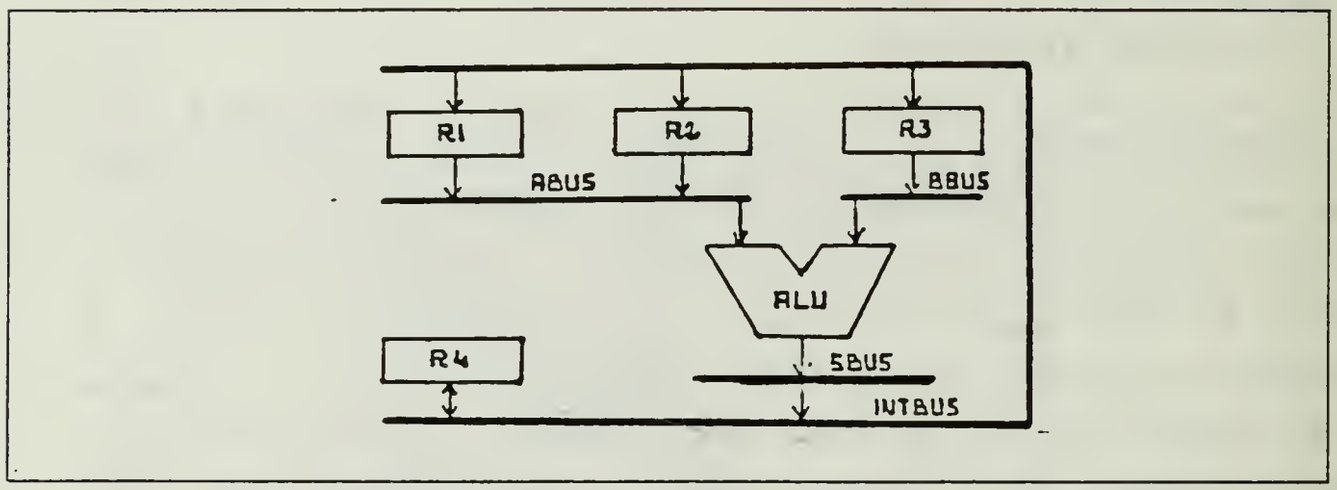


Figure 5.4 Data Paths Including Functional Units.

$$ABUS \leftarrow R1$$
$$SBUS \leftarrow (+1)ABUS$$
$$INTBUS \leftarrow SBUS$$
$$R1 \leftarrow INTBUS$$

Rule 2 applies on SBUS, simplifying the above transfer steps into:

```
ABUS ← R1              :readR1
INTBUS ← (+1)ABUS      :increment
R1 ← INTBUS            :loadR1
```

Suppose now that a binary operation is wanted, for example, add the content of R2 with the content of R3 and store the result in R4. The transfer steps needed to carried out the intended functional transfer are:

```
ABUS ← R2
BBUS ← R3
SBUS ← (+)ABUS,BBUS
INTBUS ← SBUS
R4 ← INTBUS
```

Applying Rule 2, the above set of transfer steps reduces to:

```
ABUS ← R2                 :readR2
INTBUS ← (+)ABUS,R3       :add
R4 ← INTBUS               :loadR4
```

Note that, when dealing with a functional transfer performing a binary operation, the transfer step involving the functional unit, called the functional transfer step, has two sources and is treated as two different transfer steps in one. In the above example, Rule 2 was applied having this in mind as illustrated below:

```
ABUS ← R2                 BBUS ← R3
SBUS ← (+)ABUS,x          SBUS ← (+)x,BBUS
                          SBUS ← (+)x,R3
```

```
ABUS ← R2
SBUS ← (+)ABUS,R3
```

C. DATA TRANSFERS USING UNITS

When Units were introduced in the last chapter, the following two rules were presented:

- The data always enters the Unit through an INBUS.
- The data always exits the Unit through an OUTBUS.

It was also stated at that time that these buses may not have physical existence, but they are used only as abstractions. This means that the first thing to do in the process of transfer steps simplification, is to eliminate the transfer steps containing these buses. This is done applying the following rule:

(1) In any data transfer, two transfer steps involving a INBUS or an OUTBUS are merged into a single transfer by eliminating the bus and keeping the source of the first transfer step and the sink of the second one.

Note that this rule must be the first one to apply when simplifying transfer steps. This is the reason why it is numbered one although it was the fourth one to be introduced. As an example, the transfer steps:

RFILOUTBUS ← RFIL
ADDR ← RFILOUTBUS
INTBUS ← ACC
RFILINBUS ← INTBUS
RFIL ← RFILINBUS

after Rule 1 being applied reduce to

ADDR ← RFIL
INTBUS ← ACC
RFIL ← INTBUS

If the INBUS or OUTBUS in question, have explicit control signals, the ones necessary to the transfer in question will be displayed in the control field of the resultant transfer step.

D. DATA TRANSFERS INVOLVING MEMORY DEVICES OTHER THAN REGISTERS

Until now only inter-register transfers were analyzed. At the level of abstraction where the present discussion is carried out, the differences between the several types of

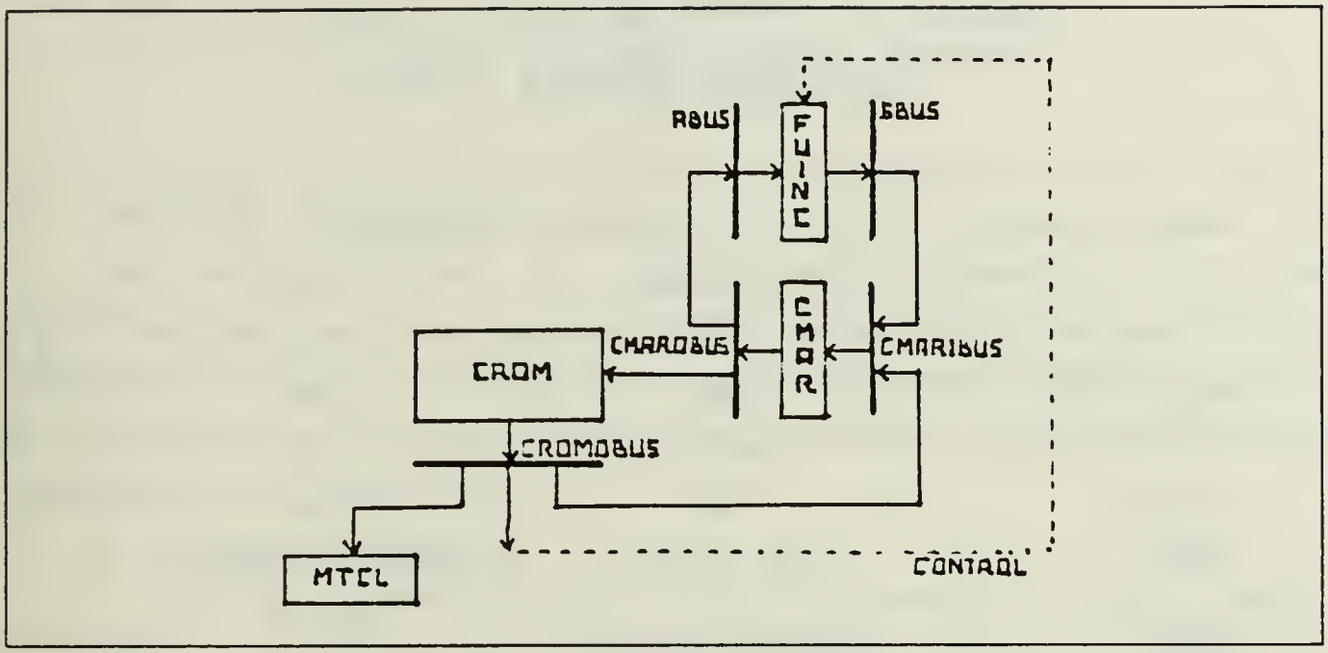


Figure 5.5 Simplified Block Diagram for a Microprogramed Control Unit.

memory devices occur in the timing constrains and the necessary control signals to operate each kind of device. Thus, the rules introduced in the last sections apply to all types of memory devices with the control signals given by Table VI.

As an example consider Figure 5.5 which shows a very simplified block diagram of the data flow of a microprogrammed control unit. The Control Memory Address Register CMAR serves as a microprogram counter. It can be incremented or loaded with the address of a microprogram subroutine, depending on the value of bits 16 and 17 of the Control ROM output vector. Its description is the following:

```
DF: MPCU;
FU: FUINC{[(),(+1),()],[]};
REG: CMAR<0:15>{[(CMARIBUS)],[(CMAROBUS)]},
     MTCL<0:32>{[(ROMOBUS<18:50>)],[]};
ROM: CROM<0:15,0:50>{[(CROMOBUS)],[(CMAROBUS)]};
BUS: ABUS<0:15>{[(CMAR)],[(FUINC)]},
     SBUS<0:15>{[(FUINC)],[(CMARIBUS•(inc))]},
```

TABLE VI

CONTROL SIGNALS FOR ALL TYPES OF MEMORY DEVICES

Memory Device	Control Signals	
	to read	to write
Registers	read	load
Stacks	push	pop
Queues	read	load
RAM's	chip select	chip select
ROM's	chip select	write

```
    CROMOBUS<0:50>{[(CROM)],[<18:50>(MTCL),
            <0:15>(CMARIBUS•(branch))]},
    CMARIBUS<0:15>{[(SBUS•(inc),
            ROMOBUS<0:15>•(branch))],[(CMAR)]},
    CMAROBUS<0:15>{[(CMAR)],[(CROM,ABUS)]};
END;
```

Suppose that bit 16 is one in present ROM output vector. Then the transfer steps necessary to produce a new output vector are the following:

```
MTCL ← CROMOBUS<18:50>
ABUS ← CMAR
SBUS ← (+1)ABUS
CMARIBUS ← SBUS
CMAR ← CMARIBUS
CROMOBUS ← CROM[CMAR]
```

Applying Rule 2, the above transfer steps simplify to:

```
MTCL ← CROMOBUS<18:50>    :loadMCTL
CMARIBUS ← (+1)CMAR       :inc
```

```
CMAR ← CMARIBUS              :loadCMAR
CROMOBUS ← CROM[CMAR]
```

E. PARALLEL TRANSFERS

The determination of which data transfers can be performed in parallel is important since parallel operation reduces the instruction time. It is therefore useful to have an easy way to determine the data transfers passive of being carried out at same time.

1. Simple Transfers

If the candidates for parallel operation are simple transfers, two cases may occur depending on the existence of just one source or more than one source for the transfers in question. For the first case the following rule applies:

(5) Simple data transfers having the same source can always be performed in parallel.

and for the latter one the rule to use is:

(6) Simple data transfers having different sources can be performed in parallel if their data paths are disjoint, e.g., if any bus is not used simultaneously by two or more transfers.

Note that when looking for parallel operation, different parts of the same device are considered to be distinct sources, sinks, or buses. For example, in the following transfers:

```
INTBUS ← A<0:7>
B ← INTBUS
C ← A<8:15>
```

have different sources, and the following transfers:

```
TOBUS<8:7> ← A
C ← TOBUS<0:7>
```

TOBUS<8:15> ← B

D ← TOBUS<0:15>

are two data transfers that can be performed in parallel because their paths are disjoint.

2. Functional Transfers

Because functional transfers use functional units that can not perform two operations at same time, the rules that apply in this type of data transfers are the following:

(7) Different functional transfers using a single functional unit can never be performed in parallel.

(8) Different functional transfers using distinct functional units, can be performed in parallel if the data paths succeeding them are disjoint, e.g., a bus can not appear simultaneously in more than one data transfer after the functional transfer step.

F. AN EXAMPLE

In order to exemplify the use of the rules introduced in this chapter, three instructions from the 8085A Microprocessor Instruction Set will be analyzed. The 8085A was presented in last chapter and its data flow description is depicted in Appendix C. Consider the instruction:

MOV r1, r2

which means that the content of register r2 is to be moved to register r1, where r1 and r2 can be any of the six general-purpose registers or the accumulator. Suppose that r2 is register B and r1 is the ACC. In order to carry out this instruction, the following simple data transfers are necessary:

RFILOUTBUS ← RFIL

INTBUS ← RFILOUTBUS<0:7>

ACC ← INTBUS

Expanding the RFIL unit, they become:

```
BOBUS ← B
RFILOUTBUS<8:15> ← BOBUS
INTBUS ← RFILOUTBUS<8:15>
ACC ← INTBUS
```

Applying Rule 1 this set becomes:

```
BOBUS ← B
INTBUS ← BOBUS    :readB
ACC ← INTBUS
```

which can be reduced by Rule 2 to:

```
INTBUS ← B
ACC ← INTBUS
```

and finally Rule 3 gives:

```
INTBUS ← B    :readB
ACC ← INTBUS  :loadACC
```

This means that the execution cycle for "MOV r1,r2" instruction needs one data transfer, e.g., it needs one clock cycle. Because in the 8085A a control state corresponds to each clock cycle, and because the fetch cycle takes three control states, the total number of states necessary to carried out this instruction are four control states.

Consider now the instruction:

```
XCHG
```

that exchanges the content of register pair HL with the content of register pair DE, in other words the content of H goes to D and the content of L goes to E. This instruction is performed with the following data transfers within the RFIL unit:

HOBUS ← H
DIBUS ← HOBUS
D ← DIBUS
LOBUS ← L
EIBUS ← LOBUS
E ← EIBUS

After Rule 2 is applied the transfers simplify to:

```
DIBUS ← H    :readH
D ← DIBUS    :loadD
EIBUS ← L    :readL
E ← EIBUS    :loadE
```

This instruction needs two simple data transfers and by Rule 6 the transfers can be performed in parallel. Thus the 'XCHG" instruction also takes four control states to execute. Consider now the instruction:

ADD r

which adds the content of register r (r being any of the general- purpose registers) with the content of ACC and places the result in ACC. Suppose that r is the register C. The transfer steps necessary to carry out this instruction are the following:

COBUS ← C
RFILOUTBUS<0:7> ← COBUS
INTBUS ← RFILOUTBUS<0:7>
TEMP ← INTBUS
BBUS ← TEMP
ACCOBUS ← ACC
ABUS ← ACCOBUS
SBUS ← (+)ABUS,BBUS
INTBUS ← SBUS
ACC ← INTBUS

which after Rule 1 becomes:

```
INTBUS ← COBUS     :readC
TEMP ← INTBUS
BBUS ← TEMP
ACCOBUS ← ACC
ABUS ← ACCOBUS
SBUS ← (+)ABUS,BBUS
INTBUS ← SBUS
ACC ← INTBUS
```

and after Rule 2 being applied, the transfers simplify to:

```
INTBUS ← C     :readC
TEMP ← INTBUS
BBUS ← TEMP
ABUS ← ACC
SBUS ← (+)ABUS,BBUS
INTBUS ← SBUS
ACC ← INTBUS
```

Applying Rule 2 again the above transfer steps become:

```
INTBUS ← C     :readC
TEMP ← INTBUS
SBUS ← (+)ACC,TEMP
INTBUS ← SBUS
ACC ← INTBUS
```

The above set of transfer steps is not yet totally simplified Rule 2 can be applied once more, giving:

```
INTBUS ← C           :readC
TEMP ← INTBUS
INTBUS ← (+)ACC,TEMP  :add
ACC ← INTBUS
```

and finally Rules 3 and 4 activate the control signals necessary to carry out the "ADD r" instruction as shown below:

```
INTBUS ← C              :readC
TEMP ← INTBUS           :loadTEMP
INTBUS ← (+)ACC,TEMP    :add
ACC ← INTBUS            :loadACC
```

As can be seen, this instruction needs one simple transfer and one functional transfer. These transfers can not be done in parallel because they both share register TEMP, which means that five control states must be paid to execute this instruction if a single-phase scheme is utilized. If a two-phase clock is used, then TEMP can be driven with one phase and ACC and C with the other phase, thereby reducing the control states necessary to perform the instruction. The Instruction Set requires only four control states for this instruction and due to the fact that the instruction is present in IR only after the third control state, the latter case must apply to the 8085A. This may be the reason why TEMP can never be operated directly by any instruction and why some 8085A block diagrams not even show it.

G. SKETCH OF A POSSIBLE WAY TO STORE THE INFORMATION CONTAINED IN THE LANGUAGE

For the information contained in the language to be useful as part of a Computer Aided Design (CAD) system, the information in the CAD database. The language introduced was designed with the Network Database Model [Ref. 13] in mind. Each type of device can be a logical record type having as fields the syntax fields of its description. For example, the register type will have the name, length, clock, source attachment and sink attachment fields. Because the last two may refer to more than one device, it is thought that the fields should be nothing more than pointers to another record type, the attachment type, comprising five fields: one for each subvector playing an active part in the attachment, one for the control signal responsible for the

transfer of data through it, one pointing to the source or sink in question and one pointing to the next attachment record of the device.

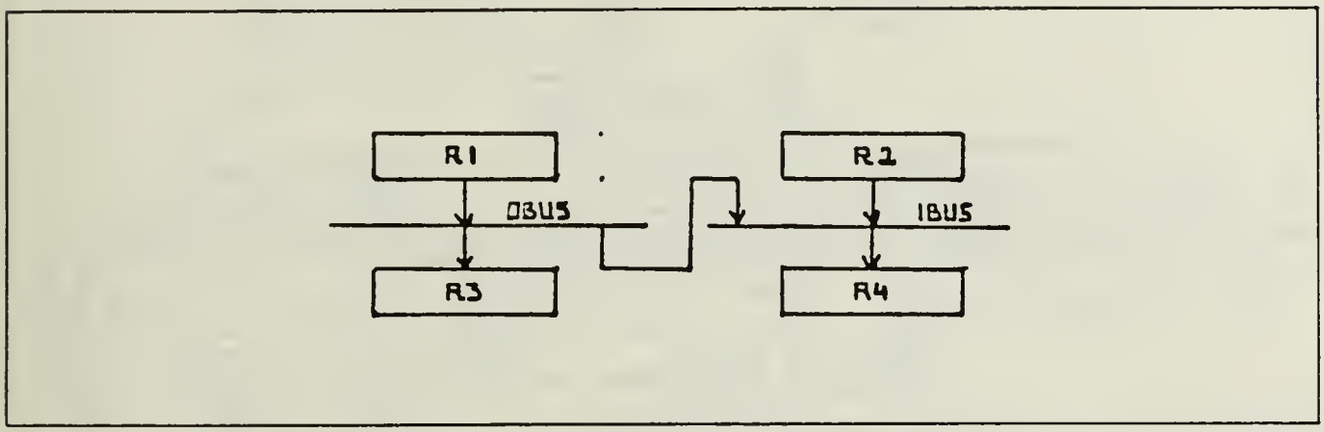


Figure 5.6 EXAMP Data Flow.

As an example consider Figure 5.6, which shows an example of a data flow whose description is the following:

```
DF: EXAMP;
REG: R0<0:7>(1,f){[],[(OBUS)]},
     R2<0:7>(1,f){[],[IBUS]},
     R3<0:3>(1,f){[(OBUS<1:3>•(ctA)],[]},
     R4<0:7>(2,f){[IBUS],[]};
BUS: OBUS<0:3>{[(R1)],[<1:3>(R3•(ctA)),(IBUS)]},
     IBUS<0:7>{[(OBUS,R2)],[R4]};
END;
```

Using the scheme presented above, this data flow will be stored as shown in Figure 5.7.

As can be seen, this scheme uses a lot of redundancy (note that the information for each attachment is stored twice), which reflects the redundancy already existing in the description itself. This drawback can be avoided by storing only one type of attachment. The reason for this comes from the fact that if all the attachments of one type (source or sink) are listed, then every connection in the data flow is known. In other words, if R is a source of B

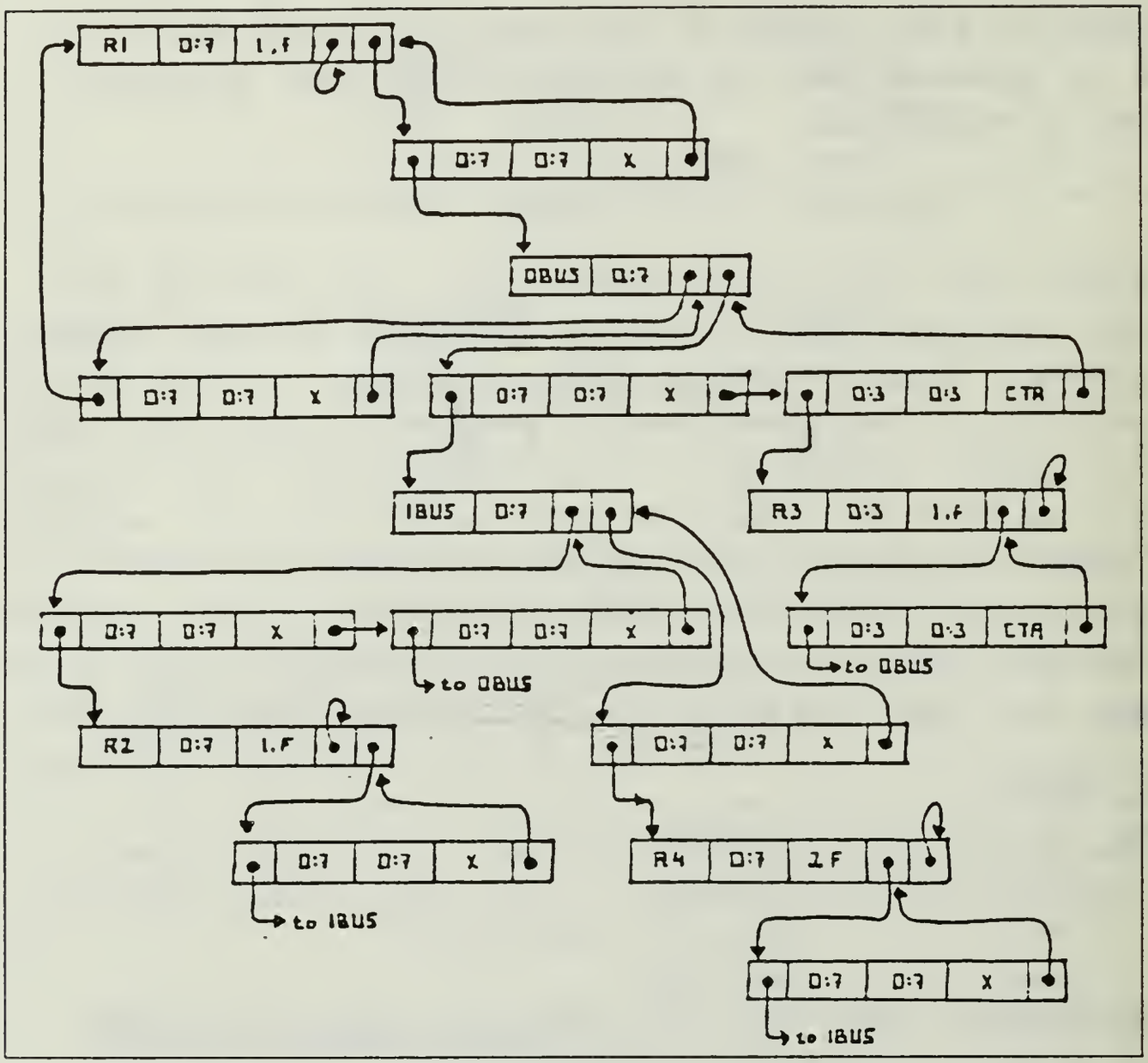


Figure 5.7 Storage of Examp.

then B is a sink for R. If this is the case, why not omit the source or the sink attachment field in the language syntax? The reason is that, a data flow described as presented in Chapters 3 and 4 is easier to be understood by the user.

In the previous example, suppose that only the sinks are stored. Then Figure 5.7 simplifies to the diagram depicted in Figure 5.8.

The possible data paths for a particular data transfer, can be obtained with a procedure that will look like:

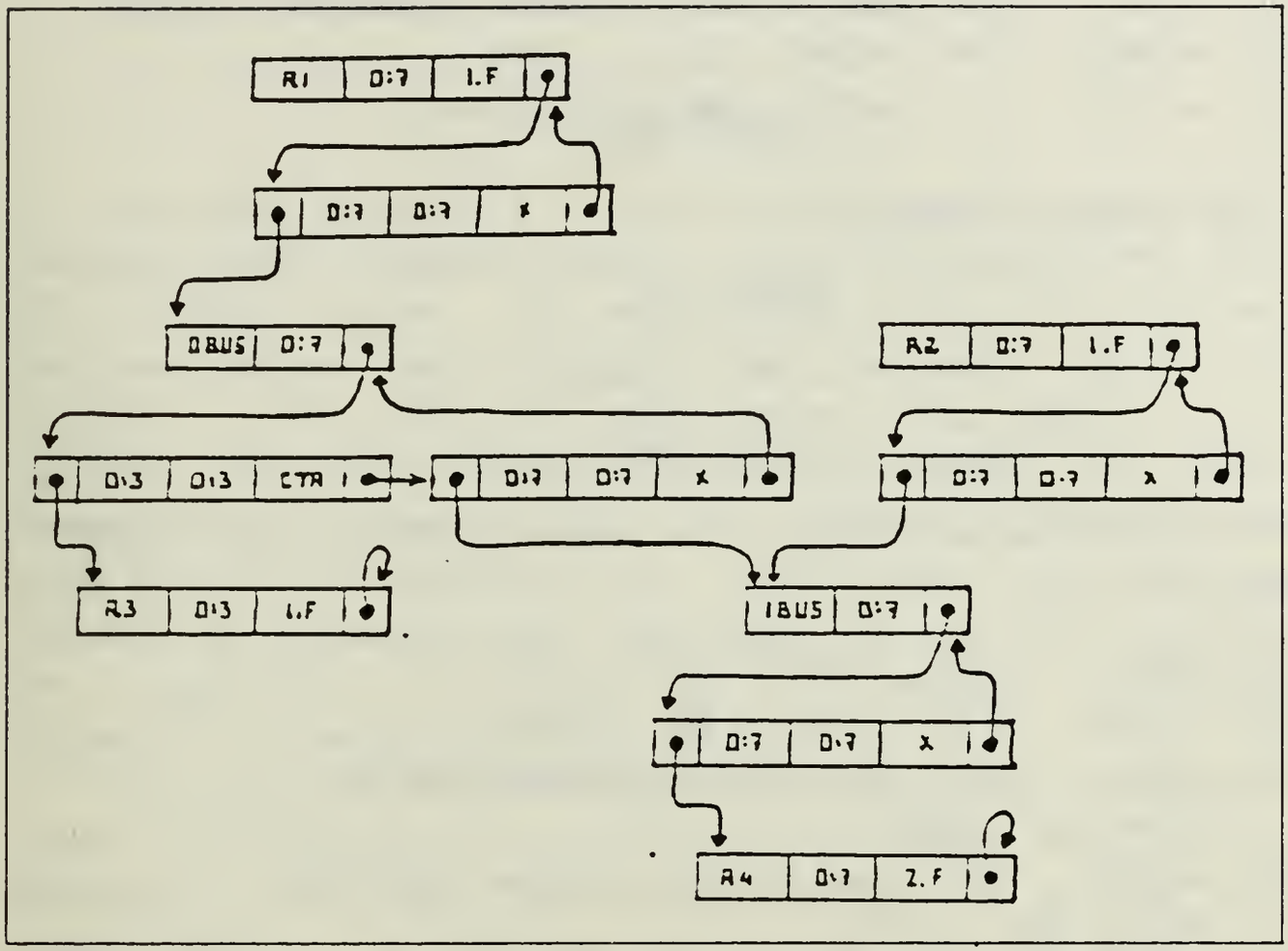


Figure 5.8 An Alternative Way to Store EXAMP.

```
   Procedure FIND(a,c)
      until end of sink list a
         pick next sink b of a
          write transfer step 'b ← a'
         if b=c, go to 10
         call FIND(b,c)
10  end
```

For a particular data transfer with source 'a' and sink 'c', procedure FIND outputs the transfer steps for all the possible data paths for the data transfer (more than one may be found). As an example, to move the contents of R1 to R4 in the EXAMP data flow, procedure FIND gives the following:

(1) a=R1 , c=R4, b=OBUS, OBUS ← R1
(2) a=OBUS, c=R4, b=R3 , R3 ← OBUS

(3) a=R3 , c=R4, end of list reached, erase R3 ← OBUS
(4) a=OBUS, c=R4, b=IBUS, IBUS ← OBUS
(5) a=IBUS, c=R4, b=R4 , R4 ← IBUS, end

The output is then:

OBUS ← R1

IBUS ← OBUS

R4 ← IBUS

If it is wanted to move the contents of R2 to R3, FIND gives:

(1) a=R2 , C=R3, b=IBUS, IBUS ← R2
(2) a=IBUS , C=R3, b=R4 , R4 ← IBUS
(3) a=R4 , C=R3, end of list reached for R4,
erase R4 ← IBUS
(4) a=IBUS , C=R3, end of list reached for IBUS,
erase IBUS ← R2
(5) end of list reached for R2, end

which means that it is not possible to move the contents of R2 to R3.

VI. CONCLUSION

As stated in Chapter 1, the objective of this work is to design a formal language capable of describing the Data Flow of a digital system. The description is for the interconnections between the major data flow components and for the control of the flow of information among them. It is believed that this objective has been achieved. The syntax presented in Chapters 3 and 4 is suitable to describe any data flow in an unambiguous way. Additionally the syntax is capable of describing the data paths for any data transfer in a particular data flow and the sequence of control signals to establish the data paths.

The approach taken defines the major data flow components, by defining a data flow of a system as a set of different data paths, each one starting and ending in a memory device. The simpler data paths are the ones that merely transfer the contents of one memory device to another. Some exist, however, that operate on the data flowing through them. The latter data paths include a component, called the functional unit, which is responsable for operating on the data. Interconnecting both types of data flow components are the buses. The term 'buses' was used to abstract wires, multiplexers, demultiplexers, and buses in the traditional sense.

At the level of abstraction treated in this work, it was found that memory devices can be catalogued in several types. The differentiation between each type is based on:

- dimension of the device (one or two dimensional).
- clock requirements (is the device clocked or not).
- if it is read-write or read-only.

The first criterion differentiates registers from stacks and queue memories, the second criterion distinguishes these devices from RAM's and ROM's and finally the last criterion differentiates RAM's from ROM's.

The functional unit was modelled as a box performing logic or arithmetic operations on one or both inputs. The operations are selected by issuing an appropriate vector to the functional unit control input. This component may also provide information about the status of the output by means of a status vector.

The syntax to describe each of these data flow components is presented in Chapter 3. The syntax to describe the data flow as an interconnection of these basic components is introduced in Chapter 4. It was found that it is sometimes useful to aggregate part of the components of a particular data flow in groups, called Units, with the purpose of simplifying the model. The Units were defined in Chapter 4. Because the unit is a data flow component as are memory devices and functional units, the data flow description may include more than one level of abstraction.

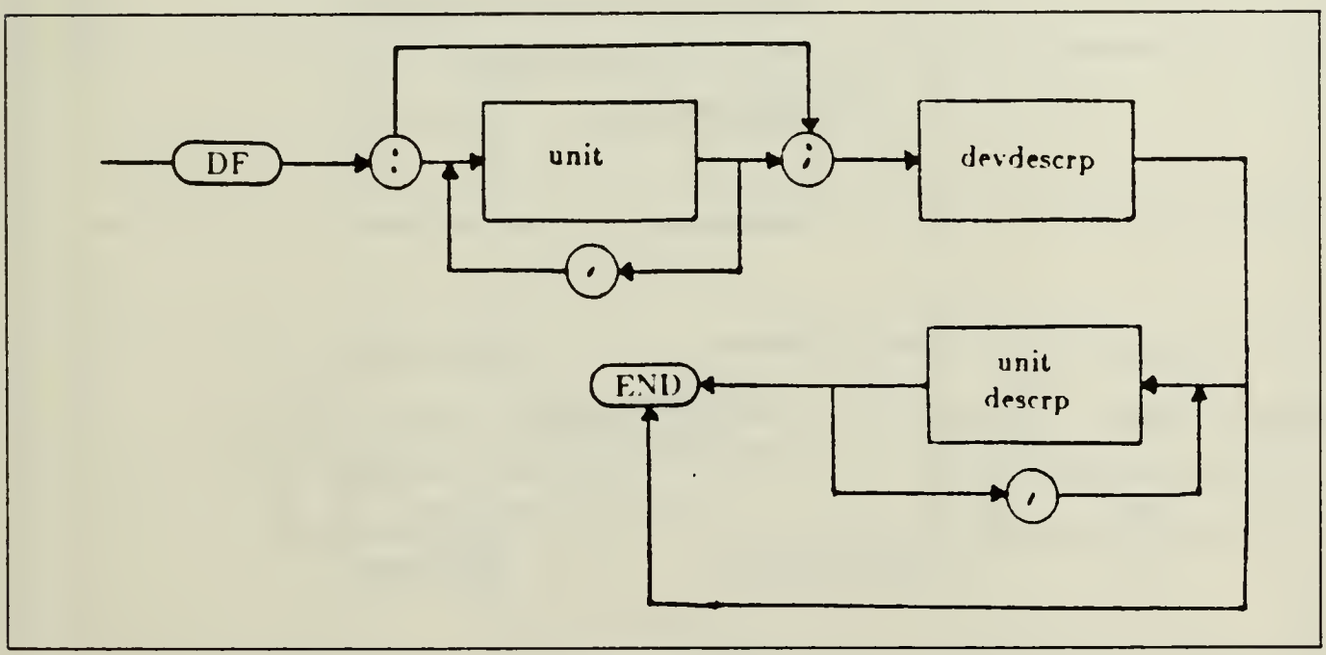
Chapter 6 showed how it is possible to obtain, from a data flow description, the control signals necessary to establish the data paths for a particular data transfer. Chapter 6 also outlined one possible way to store the information contained in the language introduced, and an algorithm to find, from a data flow description, all the possible data paths for a particular data transfer. These data paths are specified by their transfer steps, and therefore, the control signals necessary to establish the data paths can be obtained by applying the rules presented in this chapter.

APPENDIX A
SYNTAX FLOW DIAGRAMS

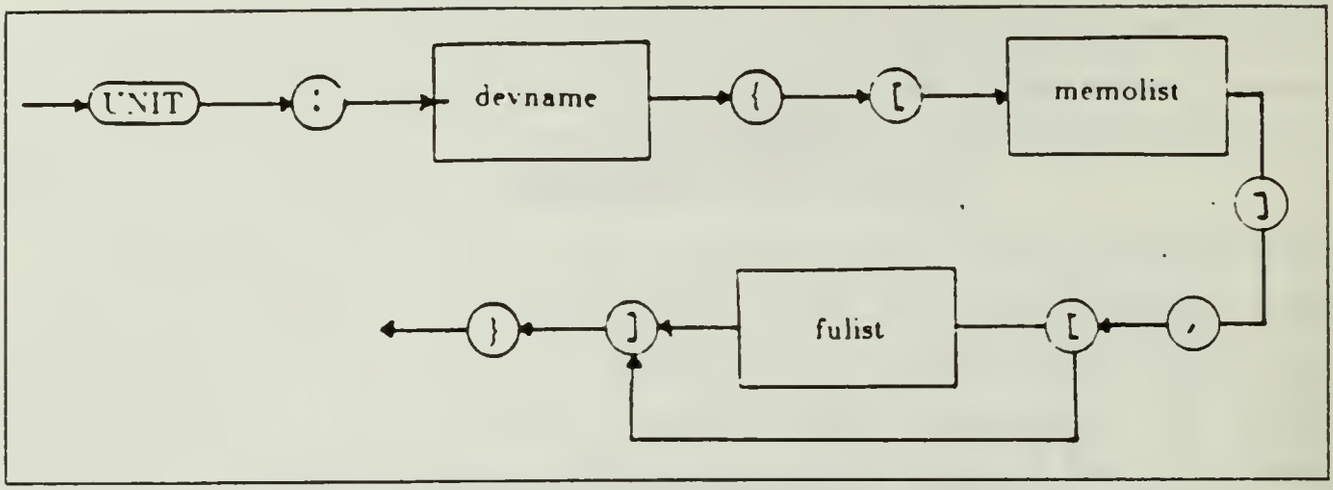
Represents reserved words or syntactic entities that are not defined further (e.g., a letter or a digit).

Represent an operator

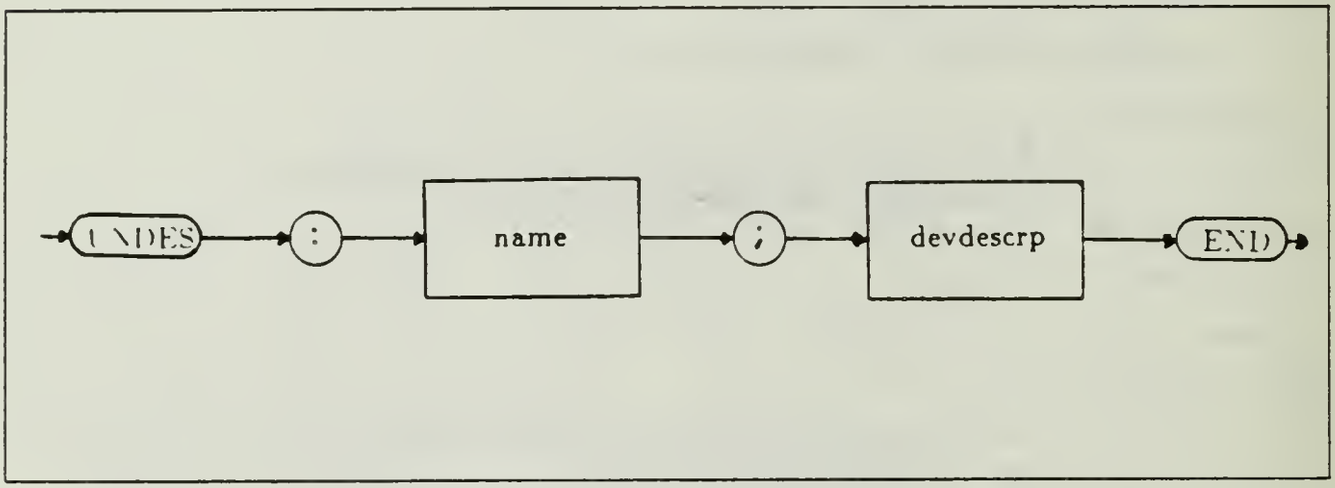
Represents a syntactic entity that is defined by another flow diagram or a table



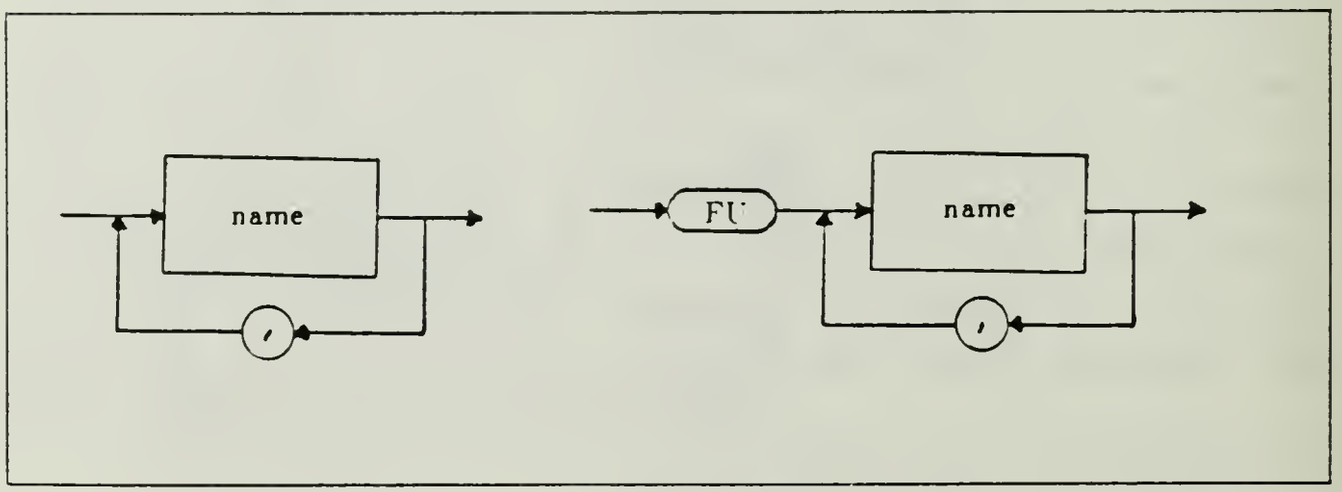
data flow



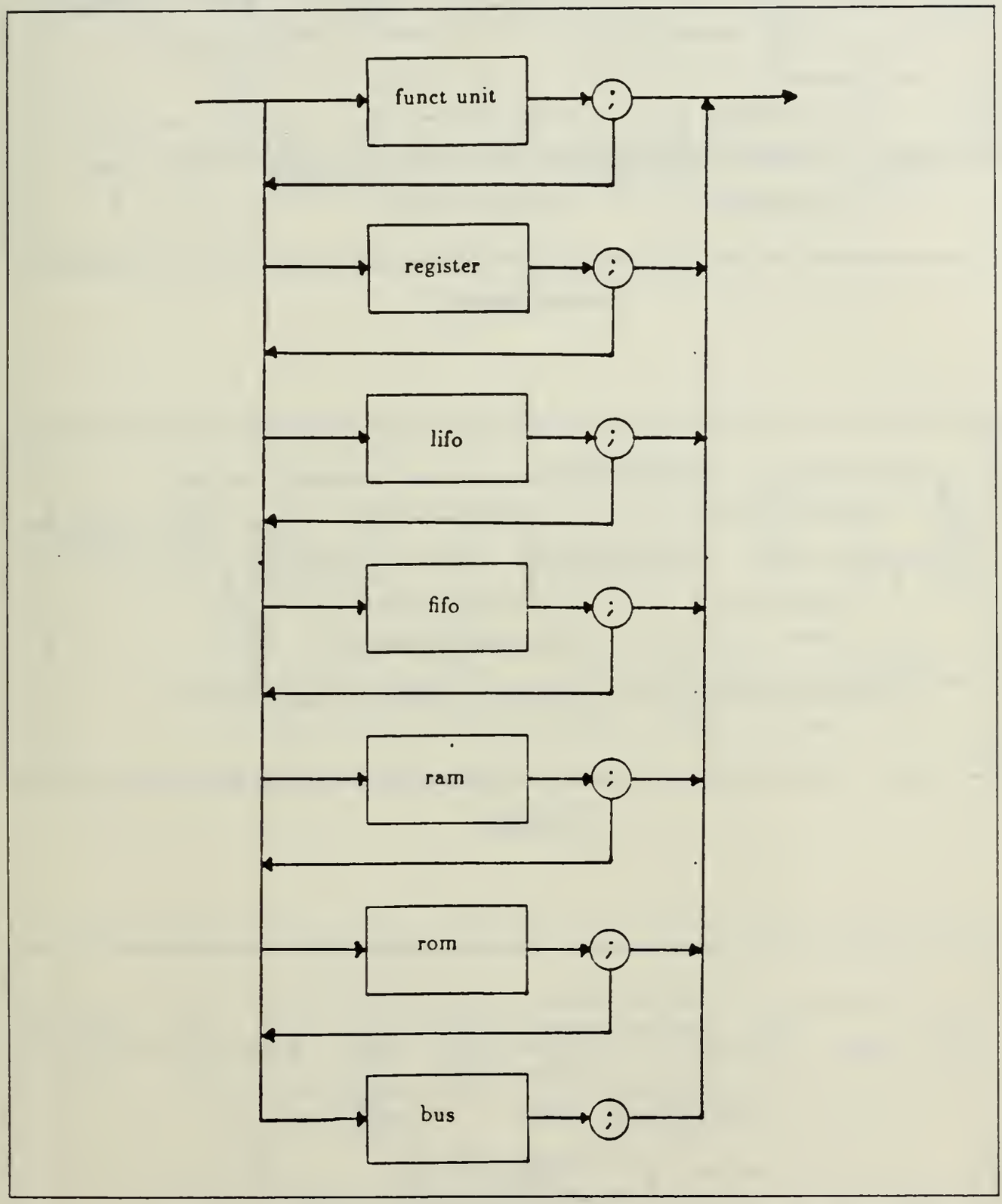
unit



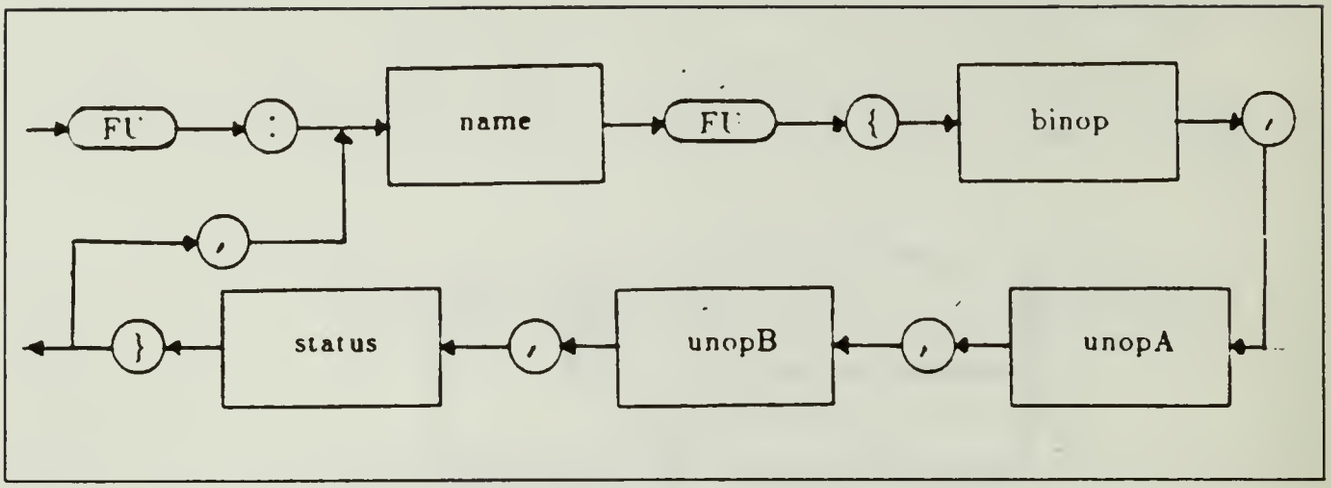
unitdescrp



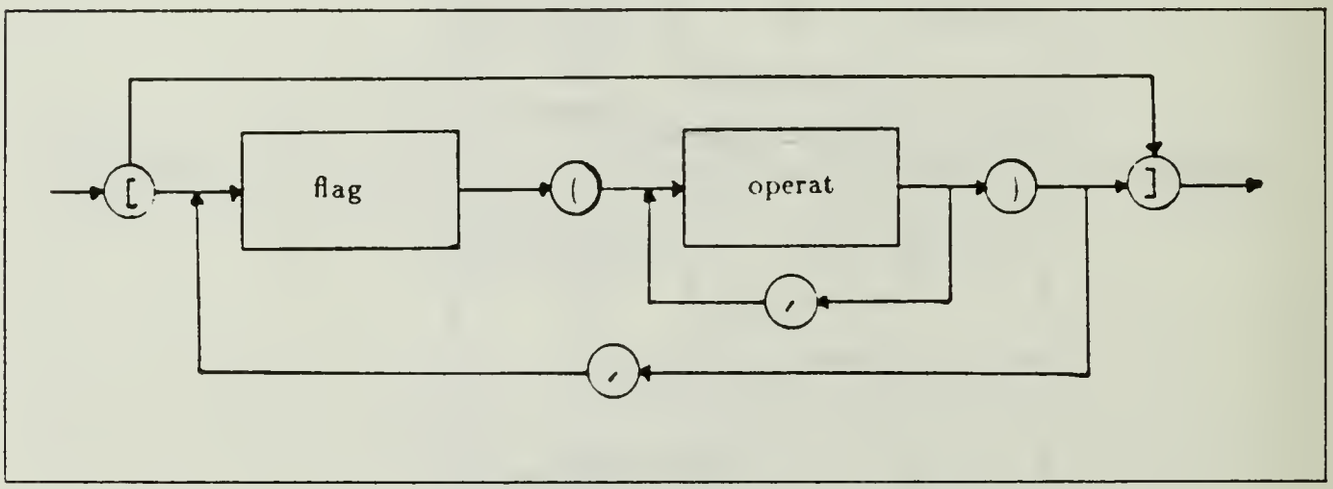
memolist fulist



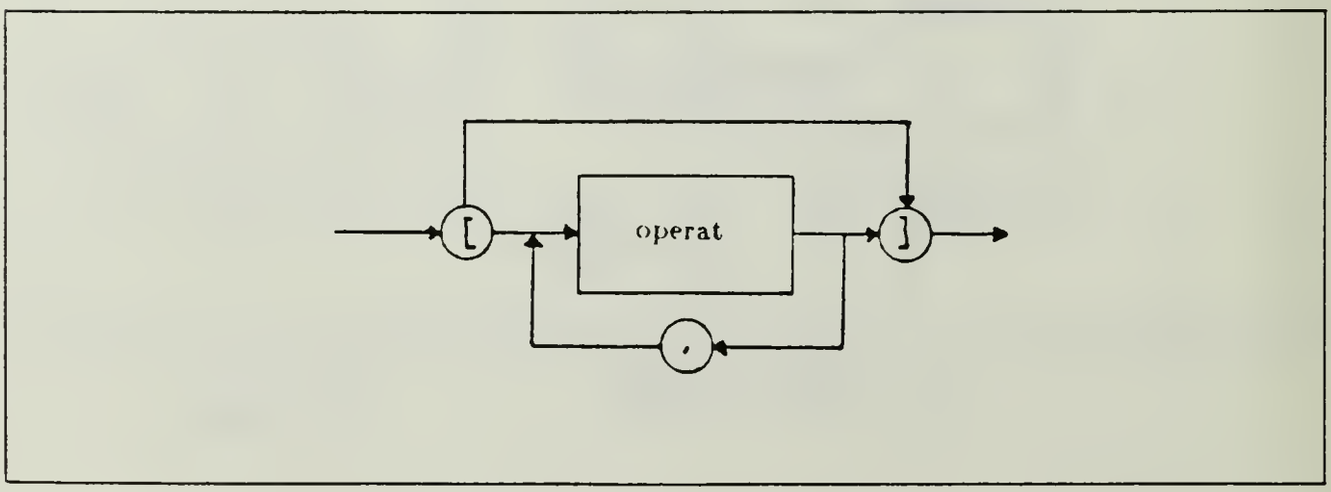
dev descrp



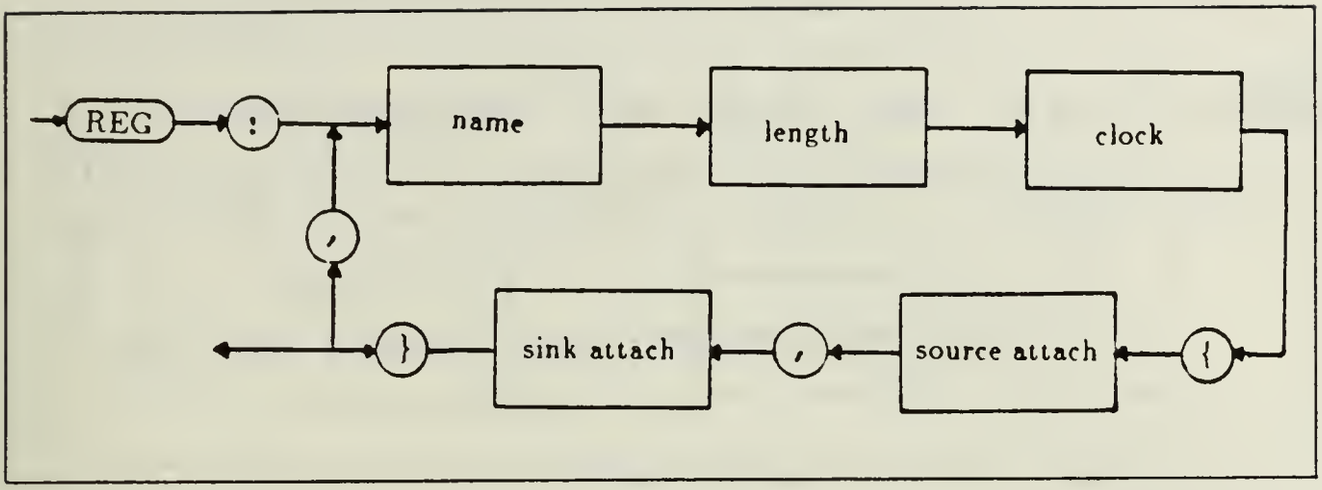
funct unit



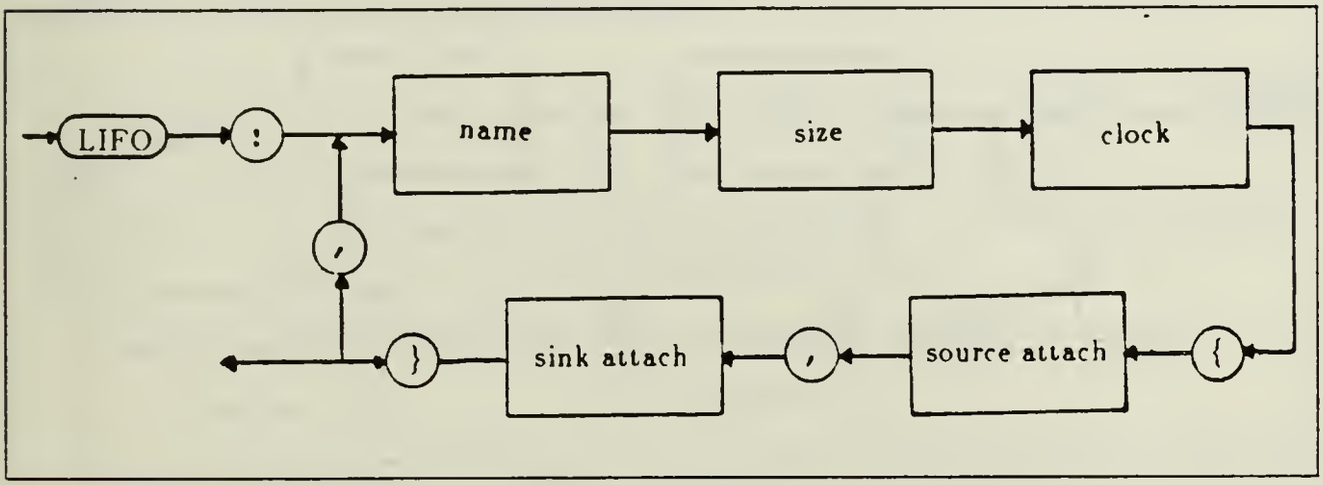
status



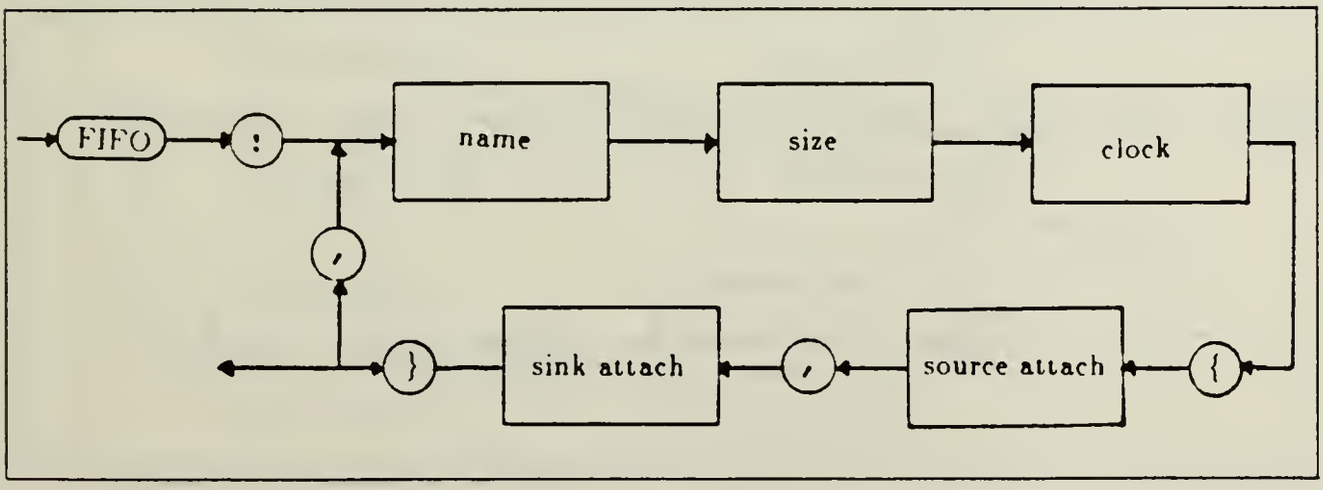
binop, unopA and unopB



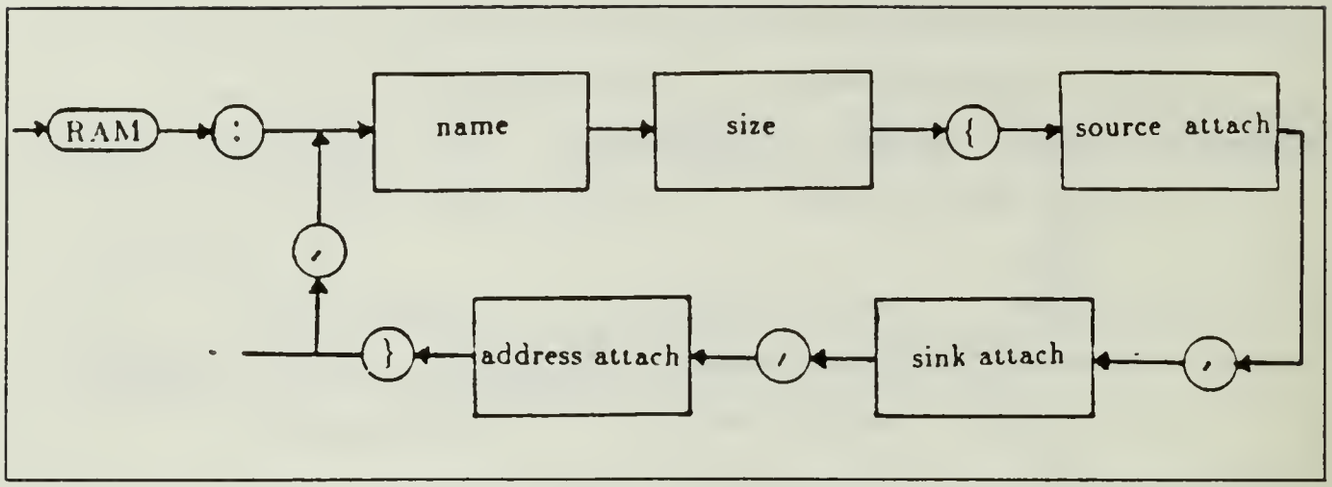
register



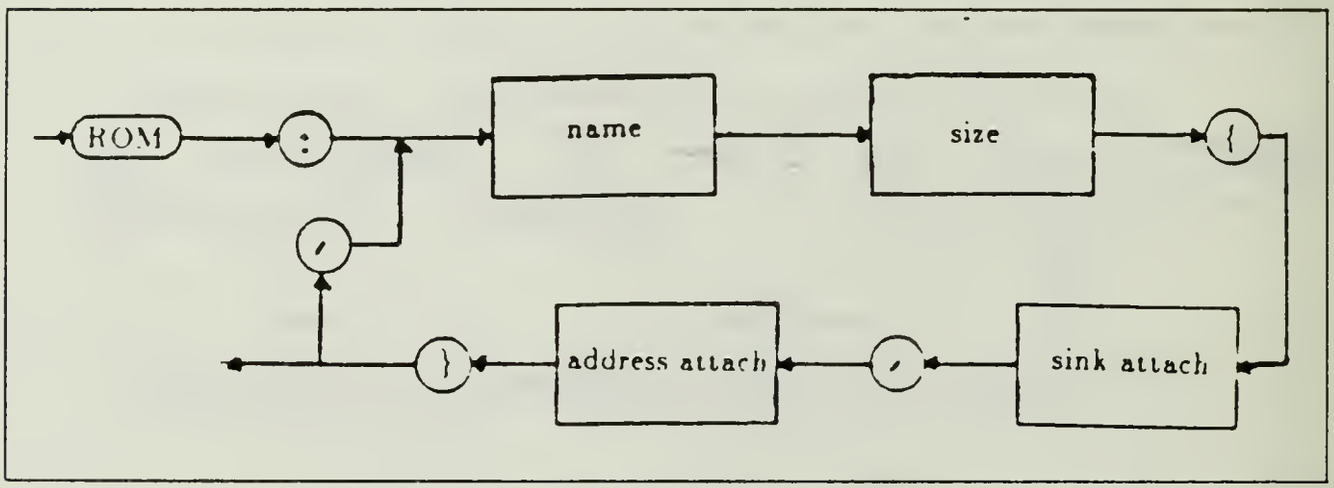
lifo



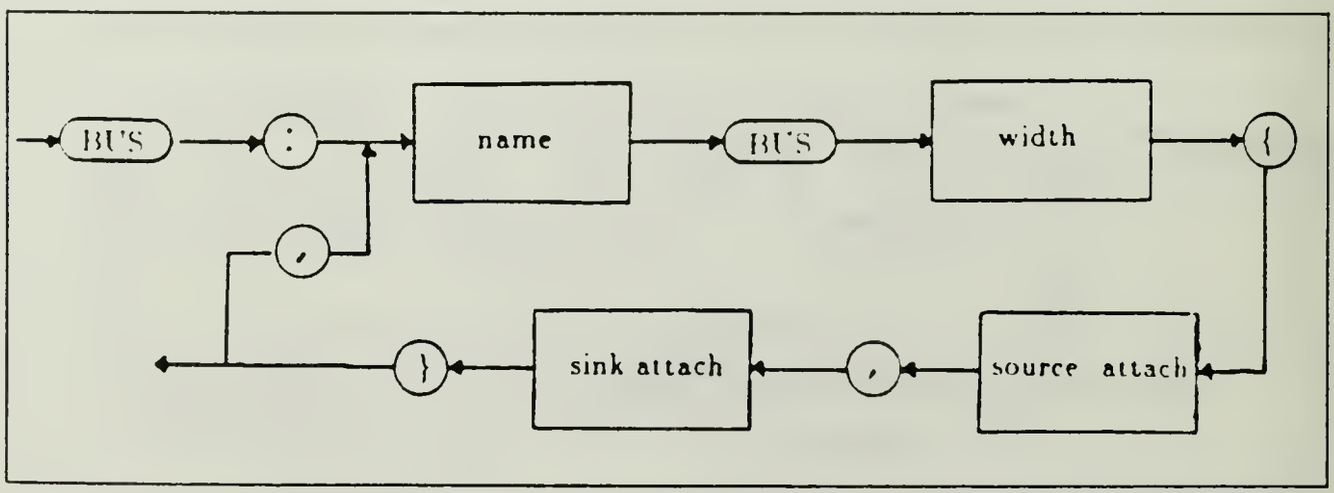
fifo


RAM
:
name
size
{
source attach
,
}
address attach
,
sink attach
,

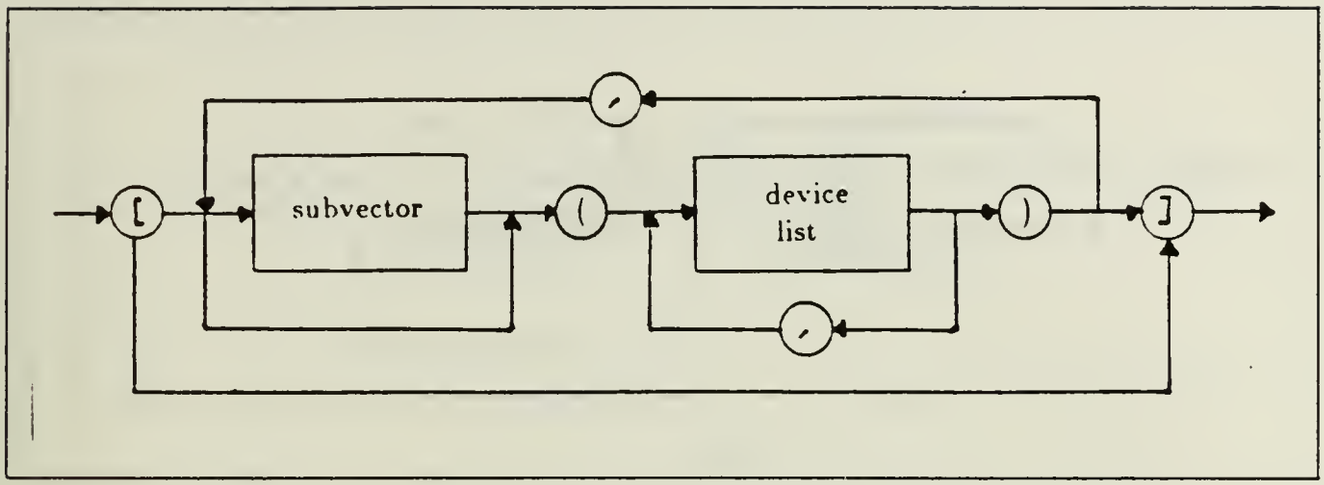
ram


ROM
:
name
size
{
,
}
address attach
,
sink attach

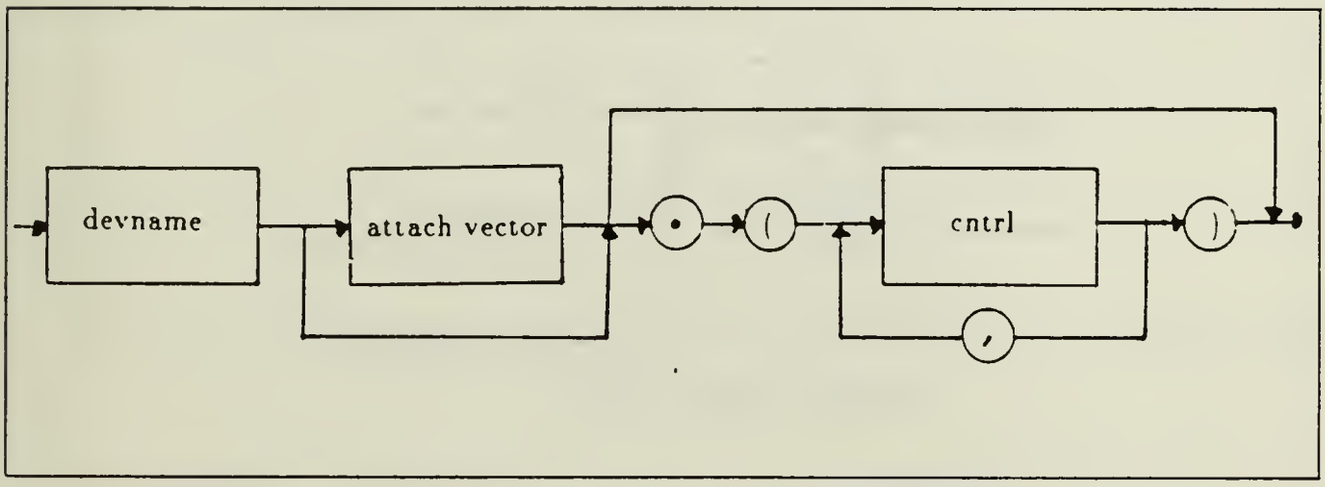
rom


BUS
:
name
BUS
width
{
,
}
sink attach
,
source attach

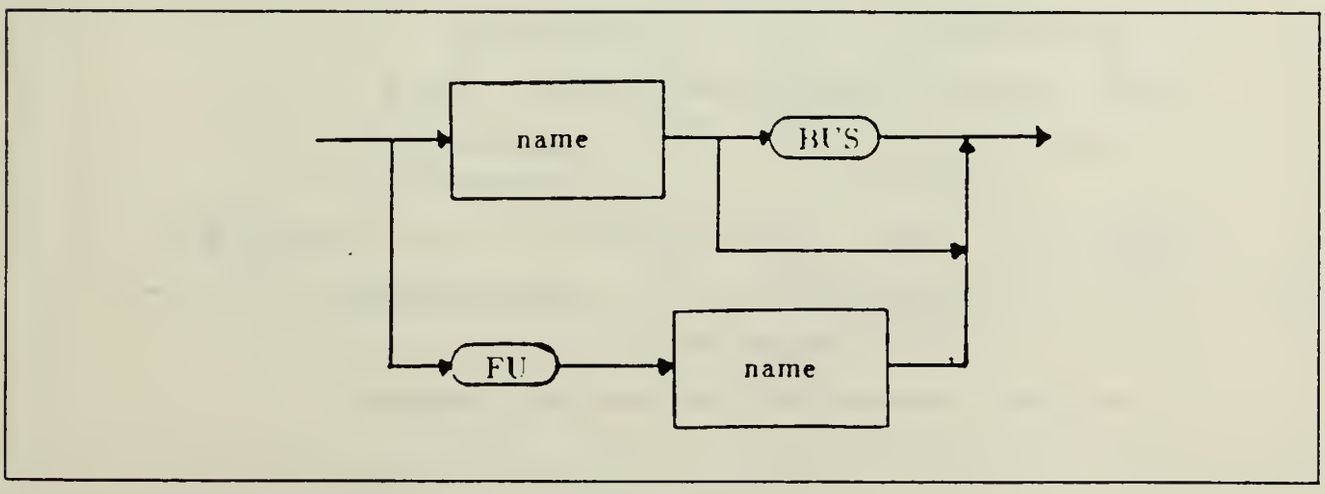
bus



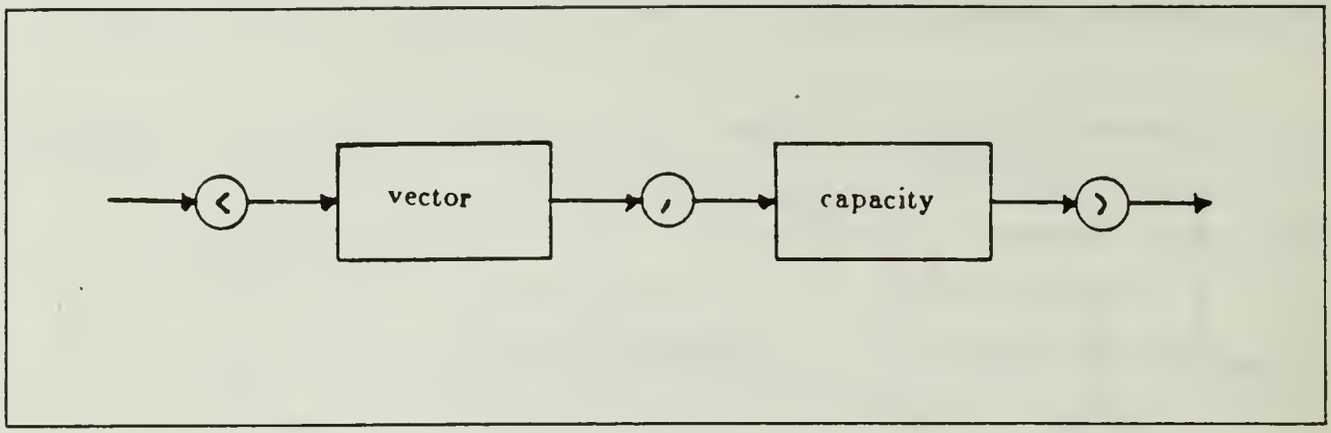
source attach, sink attach and address attach



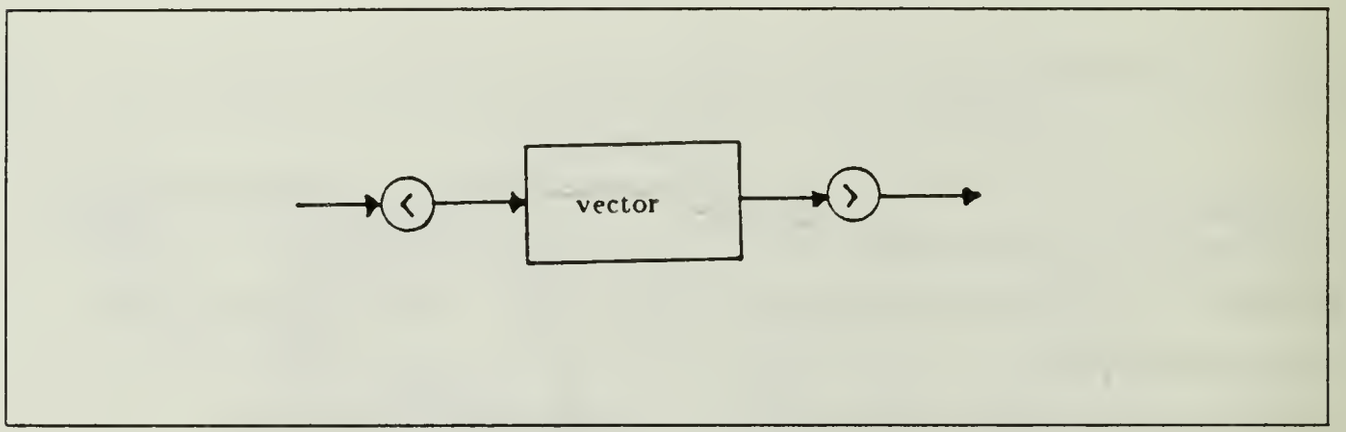
device list



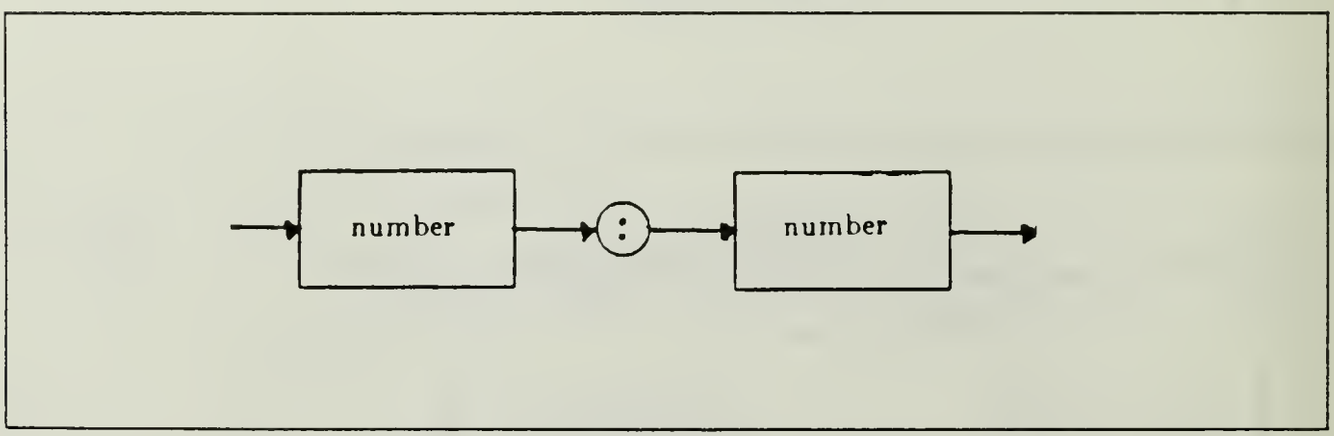
devname



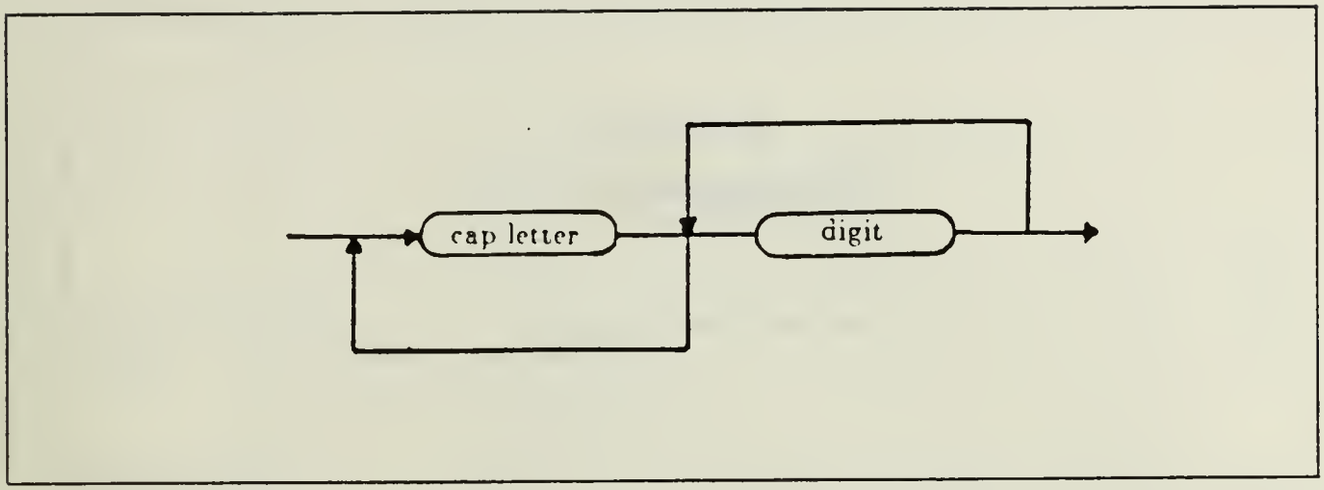
size



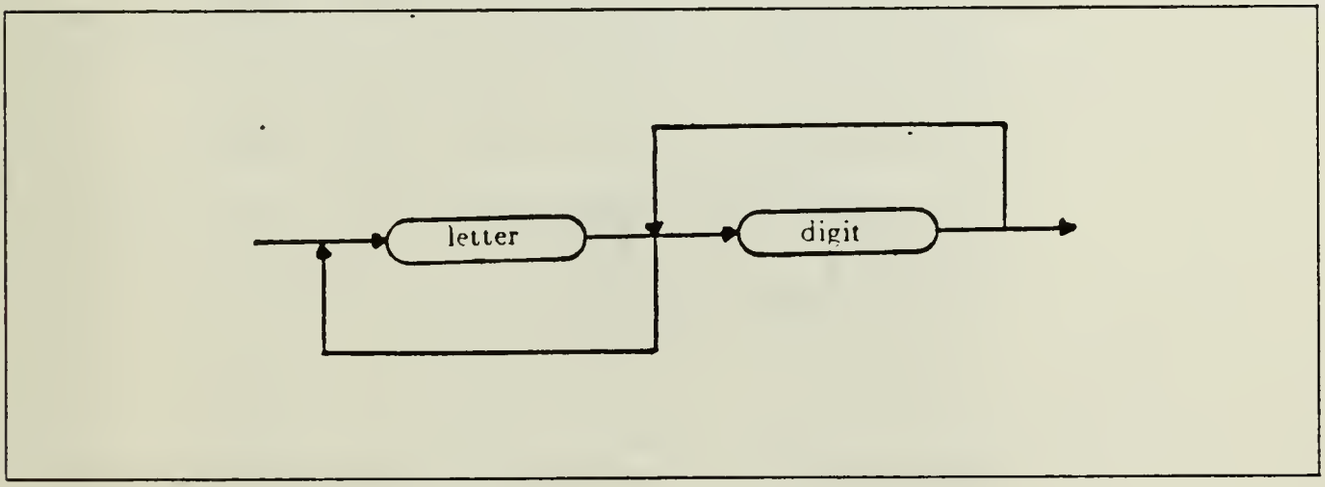
length and width



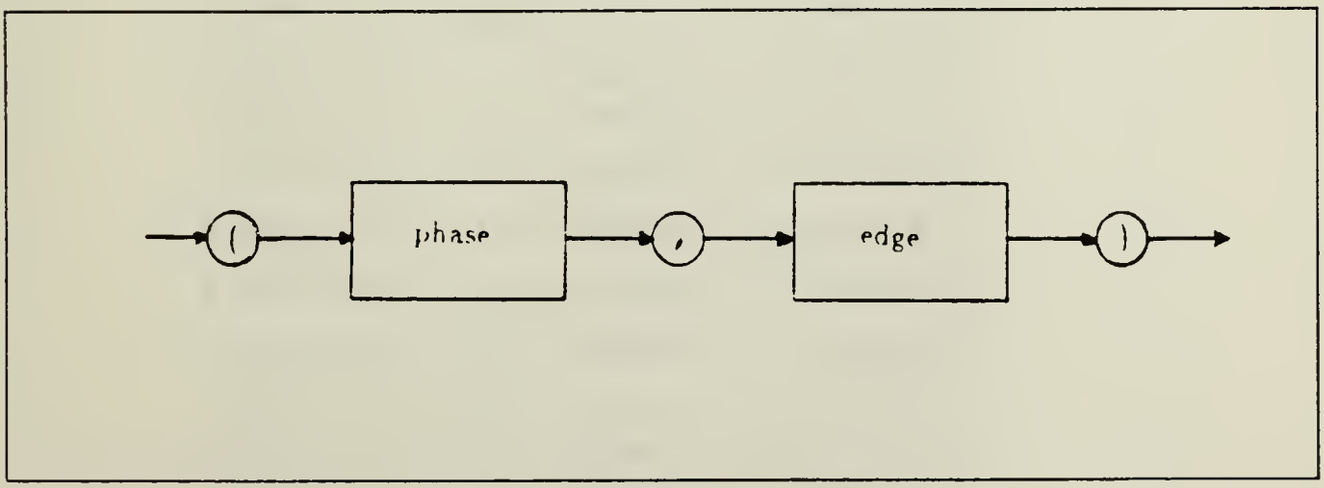
capacity,
vector, subvector and attach vector



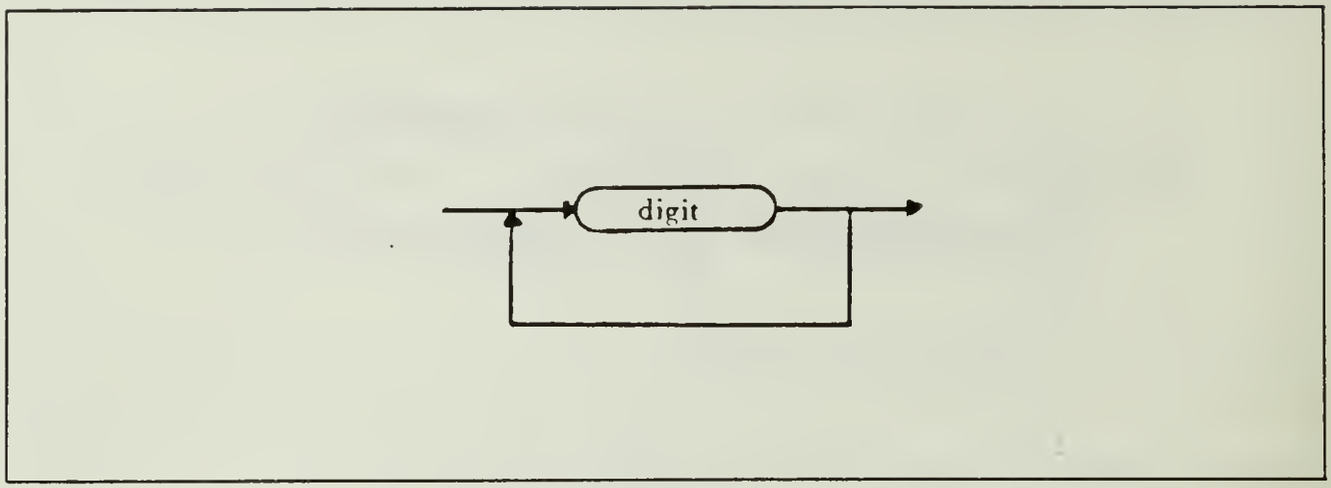
name



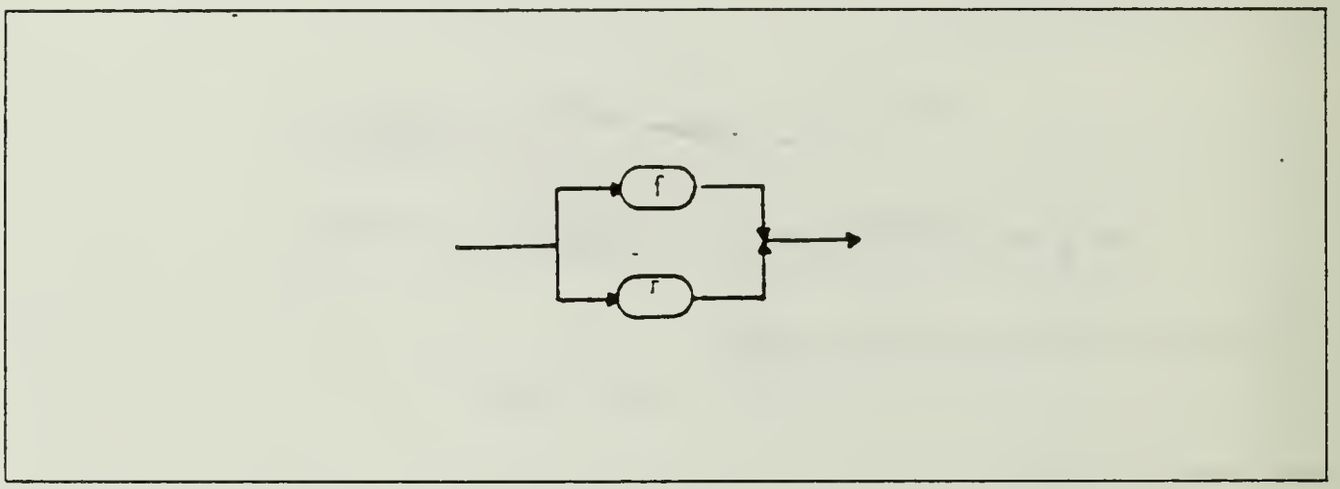
cntrl



clock



number and phase



edge

APPENDIX B
PIC 1650 DESCRIPTION

```
DF: PIC1650;
UNIT: PC{[PCREG],[FUINCA]};
      F1{[RTCC],[FUINCB]};
FU: FUALU{[(+,-,∨,⊕,∧),(0),(0,~,→,←,ℵ,+1,-1,],
      [Z(+,-,∧,∨,⊕,0,~,+1,-1),C(→,←,ℵ,+,-,),A(+,−)]}
REG: F3<0:2>(2,r){[(INTBUS<2:2>)],[(INTBUS<0:2>)]},
     F4<0:4>(2,r){[(INTBUS<2:4>)],[(INTBUS<0:4>)]},
     F5<0:7>(2,r){[(F5IBUS)],[(F5OBUS)]},
     F6<0:7>(2,r){[(F6IBUS)],[(F6OBUS)]},
     F7<0:7>(2,r){[(F7IBUS)],[(F7OBUS)]},
     F8<0:7>(2,r){[(F8IBUS)],[(F8OBUS)]},
     F9<0:7>(2,r){[(INTBUS)],[(INTBUS)]},
     F10<0:7>(2,r){[(INTBUS)],[(INTBUS)]},
     F11<0:7>(2,r){[(INTBUS)],[(INTBUS)]},
     F12<0:7>(2,r){[(INTBUS)],[(INTBUS)]},
     F13<0:7>(2,r){[(INTBUS)],[(INTBUS)]},
     F14<0:7>(2,r){[(INTBUS)],[(INTBUS)]},
     F15<0:7>(2,r){[(INTBUS)],[(INTBUS)]},
     F16<0:7>(2,r){[(INTBUS)],[(INTBUS)]},
     F17<0:7>(2,r){[(INTBUS)],[(INTBUS)]},
     F18<0:7>(2,r){[(INTBUS)],[(INTBUS)]},
     F19<0:7>(2,r){[(INTBUS)],[(INTBUS)]},
     F20<0:7>(2,r){[(INTBUS)],[(INTBUS)]},
     F21<0:7>(2,r){[(INTBUS)],[(INTBUS)]},
     F22<0:7>(2,r){[(INTBUS)],[(INTBUS)]},
     F23<0:7>(2,r){[(INTBUS)],[(INTBUS)]},
     F24<0:7>(2,r){[(INTBUS)],[(INTBUS)]},
     F25<0:7>(2,r){[(INTBUS)],[(INTBUS)]},
     F26<0:7>(2,r){[(INTBUS)],[(INTBUS)]},
     F27<0:7>(2,r){[(INTBUS)],[(INTBUS)]},
```

```
      F28<0:7>(2,r){[(INTBUS)],[(INTBUS)]},
      F29<0:7>(2,r){[(INTBUS)],[(INTBUS)]},
      F30<0:7>(2,r){[(INTBUS)],[(INTBUS)]},
      WREG<0:7>(2,r){[(INTBUS)],[(ABUS)]},
      IR<0:11>(2,r){[(OROMBUS)],[]};
LIFO: RETST<0:8,0:1>(2,r){[(PC)],[(PC)]};
ROM:  PRGM<0:11,0:511>{[(OROMBUS)],[(PC)]};
BUS:  ABUS<0:7>{[(WREG)],[(ALU)]},
      BBUS<0:7>{[(INTBUS)],[(ALU)]},
      SBUS<0:7>{[(ALU)],[(INTBUS)]},
      INTBUS<0:7>{[(SBUS),<0:2>(F3),<0:4>(F4,F9,F10,
            F11,F12,F13,F14,F15,F16,F17,F18,F19,
            F20,F21,F22,F23,F24,F25,F26,F27,F28,
            F29,F30,F31,F5OBUS,F6OBUS,F7OBUS,PC,
            F8OBUS,F1)],[(BBUS),<0:2>(F3),<0:4>(F4,
            WREG,F5IBUS,F6IBUS,F7IBUS,F8IBUS,F9,
            F10,F11,F12,F13,F14,F15,F16,F17,F18,
            F19,F20,F21,F22,F23,F24,F25,F26,F27,
            F28,F29,F30,F31,F1,PC)]},
      F5IBUS<0:7>{[(INTBUS,RABUS)],[(F5)]},
      F5OBUS<0:7>{[(F5)],[(INTBUS,RABUS)]},
      F6IBUS<0:7>{[(INTBUS,RBBUS)],[(F6)]},
      F6OBUS<0:7>{[(F6)],[(INTBUS,RABUS)]},
      F7IBUS<0:7>{[(INTBUS,RCBUS)],[(F7)]},
      F7OBUS<0:7>{[(F7)],[(INTBUS,RCBUS)]},
      F8IBUS<0:7>{[(INTBUS,RDBUS)],[(F8)]},
      F8OBUS<0:7>{[(F8)],[(INTBUS,RABUS)]},
      RABUS<0:7>{[(F5)],[(F5)]},
      RBBUS<0:7>{[(F6)],[(F6)]},
      RCBUS<0:7>{[(F7)],[(F7)]},
      RDBUS<0:7>{[(F8)],[(F8)]},
      PCINBUS<0:8>{[<0:7>(INTBUS,RETST)],[(PC)]},
      PCOUTBUS<0:8>{[(PC)],[<0:7>(INTBUS,RETST)]},
      F1INBUS<0:7>{[(INTBUS)],[(F1)]},
      F1OUTBUS<0:7>{[(F1)],[(INTBUS)]};
```

```
UNDES: PC;
FU: FUINCA{[(),( ),()],[]};
REG: PCREG<0:8>(1,r){[(PCRIBUS)],[(PCROBUS)]},
BUS: ABUS<0:8>{[(PCROBUS)],[(INCA)]},
     SBUS<0:8>{[(INCA)],[(PCRIBUS)]},
     PCRIBUS<0:8>{[(PCINBUS,SBUS)],[(PCREG)]},
     PCOBUS<0:8>{[(PCREG)],[(PCOUTBUS,ABUS)]};
END;

UNDES: F1;
FU: FUINCB{[(),( ),()],[]};
REG: RTCC<0:7>(2,r){[(RTCCIBUS)],[(RTCCOBUS)]};
BUS: ABUS<0:7>{[(RTCCOBUS)],[(INCB)]},
     SBUS<0:7>{[(INCB)],[(RTCCIBUS)]},
     RTCCIBUS<0:7>{[(F1INBUS,SBUS)],[(RTCC)]},
     RTCCOUTBUS<0:7>{[(RTCC)],[(F1OUTBUS,ABUS)]};
END;
END;
```

APPENDIX C
INTEL 8085A DESCRIPTION

```
DF: INT85A;
UNIT: REIL{[BC,DE,HL,SP,PC],[INCR]};
FU: FUALU{[(+,+c,-,-b,⊕,∨,∧),(+1,-1,~,◊,
          ←,→,→c,c←),()],
          [C(+,+c,-,-b,∧,∨,⊕,◊,→,←,→c,c←),
          Z(+,+c,-,-b,+1,-1,∧,∨,⊕,◊),
          S(+,+c,-,-b,+1,-1,∧,∨,⊕,◊),
          P(+,+c,-,-b,+1,-1,∧,∨,⊕,◊),
          A(+,+c,-,-b,+1,-1,∧,∨,⊕,◊)]}
REG: ACC<0:7>(1,f){[(INTBUS)],[(ACCOBUS)]},
     TEMP<0:7>(2,f){[(INTBUS)],[(BBUS)]},
     STA<0:7>(1,f){[(INTBUS)],[(INTBUS)]},
     IR<0:7>(1,f){[(INTBUS)],[]},
     ADB<0:7>(1,f){[(REILOUTBUS<8:15>)],[(ADDBUS<8:15>)]},
     DAB<0:7>(1,f){[(DABIBUS)],[(DABOBUS)]},
     INTC<0:7>(1,f){[(INTBUS)],[(INTBUS)]},
BUS: ABUS<0:7>{[(ACCOBUS)],[(ALU)]},
     BBUS<0:7>{[(TEMP)],[(ALU)]},
     SBUS<0:7>{[(ALU)],[(INTBUS)]},
     ACCOBUS<0:7>{[(ACC)],[(ABUS,INTBUS)]},
     INTBUS<0:7>{[(RFILOUTBUS<8:15>•(readB,readD,readH,
            readSPH,readPCH),RFILOUTBUS<0:7>•(readC,readE,
            readL,readSPL,readPCL),ACCOBUS,INTC,STA,DAB)],
            [(ACC,TEMP,RFILINBUS,STA,IR,DABIBUS)]},

     DABIBUS<0:7>{[(RFILOUTBUS<0:7>,INTBUS)],[(DAB)]},
     DABOBUS<0:7>{[(DAB)],[(ADDBUS<0:7>,INT)]},
     ADDBUS<0:15>{[(<0:7>(DABOBUS),<8:15>ADB)],[]},
     RFILOUTBUS<0:15>{[(RFIL)],[<0:7>(INTBUS•(readC,
         readE,readL,readSPL,readPCL),DABIBUS),<8:15>(ADB,
         INTBUS•(readB,readD,readH,readSPH,readPCH))]},
```

```
     RFILINBUS<0:7>{[(INTBUS)],[RFIL]};

UNDES: RFIL;
FU: FUINCR{[],[+1,-1],[]};
REG: B<0:7>(1,f){[(BIBUS)],[(BOBUS)]},
     C<0:7>(1,f){[(CIBUS)],[(COBUS)]},
     D<0:7>(1,f){[(DIBUS)],[(DOBUS)]},
     E<0:7>(1,f){[(EIBUS)],[(EOBUS)]},
     H<0:7>(1,f){[(HIBUS)],[(HOBUS)]},
     L<0:7>(1,f){[(LIBUS)],[(LOBUS)]},
     PCH<0:7>(1,f){[(PCHIBUS)],[(PCHOBUS)]},
     PCL<0:7>(1,f){[(PCLIBUS)],[(PCLBOBUS)]},
     SPH<0:7>(1,f){[(SPHIBUS)],[(SPHOBUS)]},
     SPL<0:7>(1,f){[(SPLIBUS)],[(SPLOBUS)]};
BUS: ABUS<0:15>{[<0:7>(COBUS,EOBUS,LOBUS,PCLOBUS,
                  SPLOBUS),<8:15>(BOBUS,DOBUS,HOBUS,
                  SPHOBUS,PCHOBUS)],[(FUINCR)]},
     SBUS<0:15>{[(FUINCR)],[<0:7>(CIBUS,EIBUS,LIBUS,
                  PCLIBUS,SPLIBUS),<8:15>(BIBUS,DIBUS,
                  HIBUS,SPHIBUS,PCHIBUS)]},
     BIBUS<0:7>{[(RFILINBUS),<8:15>(SBUS)],[(B)]},
     CIBUS<0:7>{[(RFILINBUS),<0:7>(SBUS)],[(C)]},
     DIBUS<0:7>{[(RFILINBUS,HOBUS),<8:15>(SBUS)],[(D)]},
     EIBUS<0:7>{[(RFILINBUS,LOBUS),<0:7>(SBUS)],[(E)]},
     HIBUS<0:7>{[(RFILINBUS,DOBUS),<8:15>(SBUS)],[(H)]},
     LIBUS<0:7>{[(RFILINBUS,EOBUS),<0:7>(SBUS)],[(L)]},
     PCHIBUS<0:7>{[(RFILINBUS),<8:15>(SBUS)],[(PCH)]},
     PCLIBUS<0:7>{[(RFILINBUS),<0:7>(SBUS)],[(PCL)]},
     SPHIBUS<0:7>{[(RFILINBUS),<8:15>(SBUS)],[(SPH)]},
     SPLIBUS<0:7>{[(RFILINBUS),<0:7>(SBUS)],[(SPL)]},
     BOBUS<0:7>{[(B)],[(ABUS<8:15>,RFILOUTBUS<8:15>)]},
     COBUS<0:7>{[(C)],[(ABUS<0:7>,RFILOUTBUS<0:7>)]},
     DOBUS<0:7>{[(D)],[(ABUS<8:15>,RFILOUTBUS<8:15>,
                  HIBUS)]},
     EOBUS<0:7>{[(E)],[(ABUS<0:7>,RFILOUTBUS<0:7>,
                  LIBUS)]},
```

```
        HOBUS<0:7>{[(H)],[(ABUS<8:15>,RFILOUTBUS<8:15>,
                    DIBUS)]},
        LOBUS<0:7>{[(L)],[(ABUS<0:7>,RFILOUTBUS<0:7>,
                    EIBUS)]},
        PCHOBUS<0:7>{[(RFILINBUS),<8:15>(SBUS)],[(B)]},
        PCHOBUS<0:7>{[(PCH)],[(ABUS<8:15>,
                    RFILOUTBUS<8:15>)]},
        PCHLBUS<0:7>{[(PCL)],[(ABUS<0:7>,
                    RFILOUTBUS<0:7>)]},
        SPHOBUS<0:7>{[(SPH)],[(ABUS<0:7>,
                    RFILOUTBUS<8:15>)]},
        SPLOBUS<0:7>{[(SPL)],[(ABUS<0:7>,
                    RFILOUTBUS<0:7>)]};
END;
END;
```

LIST OF REFERENCES

1. Rigas, Harriett, A Complete Automated Design System, (unpublished).

2. Mahmood, Ausif, Development of a Multi-level Logic Simulator for VLSI Systems, Washington State University, 1985.

3. Fletcher, I. William, An Engineering Approach to Digital Design, Prentice-Hall, 1980.

4. Mano, M. Morris, Computer System Architecture, Prentice Hall, 1982.

5. Langdom, Glen G. Jr., Computer Design, Computer Press, Inc., 1982.

6. Taub, Herbet, Digital Circuits and Microprocessors, McGraw Hill Book Company, 1982.

7. Mead, Carver and Conway, Lin, Introduction to VLSI Systems, Addison Wesley, 1980.

8. Weste, Neil and Eshraghian, Kamran, Principles of CMOS VLSI Design, Addison Wesley, 1985.

9. Leventhal, Lance, Introduction to Microprocessors, Prentice Hall, 1978.

10. Madnick, Stuart and Donovan, John, Operating Systems, McGraw- Hill Book Company, 1974.

11. Sieviorek, Bell and Newell, Computer Structures: Principles and Examples, McGraw Hill Book Company, 1982.

12. Short, Kenneth L., Microprocessors and Programming Logic, Prentice Hall, 1981.

13. Ullman, D. Jeffrey, Principles of Database Systems, Computer Science Press, 1985.

INITIAL DISTRIBUTION LIST

No.		Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library,Code 0142 Naval Postgraduate School Monterey, California 93943-5002	2
3.	Dr. Harriett B. Rigas Code 62 Naval Postgraduate School Monterey, California 93943	2
4.	Dr. Larry Abbott Code 63At Naval Postgraduate School Monterey, California 93943	1
5.	Dir. Serv. Instrucao e Treino Edificio do Ministerio da Marinha Rua do Arsenal 1100 Lisboa Portugal	1
6.	Luis Sousa Machado Av. D. Rodrigo da Cunha, 12 - 2 Esq Sul 1700 Lisboa Portugal	4

217506

Thesis
M1891
c.1

Machado
A language capable of describing computer architecture.